

<u>VBSCRIPT INTRINSIC (PRE-DEFINED) PROCEDURES</u>	<u>4</u>
INTERACTION FUNCTIONS	4
INPUTBOX FUNCTION	4
MSGBOX FUNCTION	6
STRING MANIPULATION FUNCTIONS.....	8
ASC, ASCB, ASVW FUNCTIONS.....	8
CHR, CHRB, CHRW FUNCTIONS.....	9
INSTR, INSTRB FUNCTIONS	10
INSTRREV FUNCTION.....	12
LCASE, UCASE FUNCTIONS	13
LEFT, LEFTB FUNCTIONS	14
RIGHT, RIGHTB FUNCTIONS.....	15
MID, MIDB FUNCTIONS	16
LEN, LENB FUNCTIONS.....	17
LTRIM, RTRIM AND TRIM FUNCTIONS	18
STRCOMP FUNCTION	18
STRING FUNCTION	19
REPLACE FUNCTION.....	20
STRREVERSE FUNCTION	22
SPACE FUNCTION.....	22
DATE/TIME MANIPULATION FUNCTIONS	23
DATE FUNCTION	23
DATEADD FUNCTION	23
DATEDIFF FUNCTION	24
DATEPART FUNCTION.....	26
DATESERIAL FUNCTION.....	27
DATEVALUE FUNCTION	28
DAY FUNCTION.....	29
HOUR FUNCTION.....	29
MINUTE FUNCTION	30
MONTH FUNCTION.....	30
MONTHNAME FUNCTION	31
NOW FUNCTION	31
SECOND FUNCTION	32
TIME FUNCTION	32
TIMER FUNCTION.....	33
TIMESERIAL FUNCTION	33
TIMEVALUE FUNCTION.....	34
WEEKDAY FUNCTION	35
WEEKDAYNAME FUNCTION	36
YEAR FUNCTION.....	36
MATH FUNCTIONS	37
ABS FUNCTION	37
ATN FUNCTION	38
COS FUNCTION	38
SIN FUNCTION	39
TAN FUNCTION	39
LOG FUNCTION	40

EXP FUNCTION.....	41
SQR FUNCTION	41
DERIVED MATH FUNCTIONS	42
INT FUNCTION	43
FIX FUNCTION	43
HEX FUNCTION	44
OCT FUNCTION	45
SGN FUNCTION	46
ROUND FUNCTION	47
RND FUNCTION.....	48
FORMAT FUNCTIONS	48
FORMATCURRENCY, FORMATNUMBER AND FORMATPERCENT FUNCTIONS	48
FORMATDATETIME FUNCTION.....	50
DATA TYPE VALIDATION FUNCTIONS	51
VARTYPE FUNCTION.....	51
TYPENAME FUNCTION	52
ISDATE FUNCTION	53
ISNUMERIC FUNCTION	54
ISEMPTY FUNCTION	55
ISNULL FUNCTION	56
ISARRAY FUNCTION	57
ISOBJECT FUNCTION.....	58
IS NOTHING EXPRESSION	59
DATA CONVERSION FUNCTIONS.....	60
CBOOL FUNCTION	60
CBYTE FUNCTION.....	61
CDATE FUNCTION.....	62
CINT FUNCTION	64
CLNG FUNCTION	65
CSNG FUNCTION.....	66
CDBL FUNCTION.....	68
CCUR FUNCTION	69
CSTR FUNCTION	70
ARRAY FUNCTIONS.....	71
ERASE STATEMENT.....	71
UBOUND FUNCTION.....	72
LBOUND FUNCTION	73
SPLIT FUNCTION	74
JOIN FUNCTION	75
FILTER FUNCTION.....	76
ARRAY FUNCTION	78
WORKING WITH OBJECTS	79
CONNECTING TO OBJECTS.....	80
CREATING AN OBJECT REFERENCE	80
CALLING METHODS	81
RETRIEVING PROPERTIES	81
GENERAL FUNCTIONS AND STATEMENTS.....	81
EVAL FUNCTION	81
RGB FUNCTION	83
GETLOCALE FUNCTION.....	84
SETLOCALE FUNCTION	84

EXECUTE STATEMENT	85
EXECUTEGLOBAL STATEMENT	87
LOADPICTURE FUNCTION	87
CREATEOBJECT FUNCTION	88
GETOBJECT FUNCTION	90
GETREF FUNCTION	93
CALL STATEMENT	94
SET STATEMENT	94
SCRIPTENGINE FUNCTION	96
SCRIPTENGINEBUILDVERSION FUNCTION	96
SCRIPTENGINEMAJORVERSION FUNCTION	96
SCRIPTENGINEMINORVERSION FUNCTION	97
DICTIONARY OBJECT.....	97
CREATING A DICTIONARY	99
Configuring Dictionary Properties.....	99
Adding Key-Item Pairs to a Dictionary.....	99
Manipulating Keys and Items in a Dictionary.....	101
DICTIONARY OBJECT, PROPERTIES AND METHODS	101
Dictionary.CompareMode Property.....	101
Dictionary.Count Property	102
Dictionary.Item Property.....	102
Dictionary.Key Property	103
Dictionary.Add Method	104
Dictionary.Exist Method	104
Dictionary.Items Method	105
Dictionary.Keys Method	106
Dictionary.Remove Method	106
Dictionary.RemoveAll Method	106
REGEXP OBJECT.....	107
CLASS OBJECT	107
THE CLASS CONSTRUCT.....	107
THE NEW STATEMENT	108
THE ME KEYWORD	108
THE CLASS VARIABLES.....	108
THE CLASS PROPERTIES	109
Property Get Statement	109
Property Let Statement.....	112
Property Set Statement.....	114
Using properties to wrap private variables.....	115
The Class Methods.....	116
The Class Events.....	117
CLASSES IN QUICKTEST	119
CLASS DEMO: DATATABLEUTIL	120
CLASS DEMO: WIN32DIRECTORY	120
FILESYSTEMOBJECT OBJECT.....	120
ERR OBJECT	120
COM OBJECTS	120
THE COM PROCESS	121
Creating a New Object	122
Server Mode	122
BINDING	122

CHOOSING A METHOD FOR BINDING TO AN AUTOMATION OBJECT	123
VERIFYING OBJECT REFERENCES	124
UNLOADING OBJECTS FROM MEMORY	124
Nothing Keyword	125
Q&A	126
DESKTOP MANAGEMENT	126
How To List Items in the Administrative Tools Folder?	126
APPENDIX 4.A	126
MsgBox Return Value Constants	126
Button Display Constants	127
Icon Display Constants	127
Default Button Constants	128
Modality Constants	128
Comparision Constants	128
Interval Settings	128
Tristate Constants	129
Color Constants	129
Date Format Constants	129
Date and Time Constants	129
String Constants	130
VarType Constants	130
WeekDay Constants	131
APPENDIX 4.B	131
Locale Id's	131

VBScript Intrinsic (Pre-Defined) Procedures

In addition to creating your own procedures for use in your scripts, you can use a number of intrinsic function procedures that are built into VBScript. These functions include string operations, conversions, math functions, time and date functions, and Boolean functions. Understanding these functions will benefit you greatly as you begin to write larger and more complex scripts.

Interaction Functions

When running automation scripts, we usually don't use interactions function of any type, because the idea of automation testing, is testing without the need of human resources, and "interaction" means, that you need a human, to make a decision, input required data, or whatever other task.

InputBox Function

Description

The **InputBox** function displays a dialog box containing a label, which prompts the user about the data you expect them to input, a text box for entering the data, an **OK** button, a **Cancel** button, and optionally, a Help button. When the user clicks **OK**, the function returns the contents of the text box.

Syntax

```
InputBox(prompt,title,default,xpos, ypos, helpfile, context)
```

Arguments

Constant	Description
<i>prompt</i>	The message in the dialog box.
<i>title</i>	The title-bar of the dialog box.
<i>default</i>	String to be displayed in the text box on loading.
<i>xpos</i>	The distance from the left side of the screen to the left side of the dialog.
<i>ypos</i>	The distance from the top of the screen to the top of the dialog box.
<i>helpfile</i>	The Help file to use if the user clicks the Help button on the dialog box.
<i>context</i>	The context number to use within the Help file specified in <i>helpfile</i> .

Notes

1. **InputBox** returns a string containing the contents of the text box from the Input-box dialog.
2. If the user clicks Cancel, a zero-length string (" ") is returned.
3. *prompt* can contain approximately 1,000 characters, including nonprinting characters like the intrinsic `vbCrLf` constant.
4. If you don't use the *default* parameter to specify a default entry for the text box, the text box is shown empty; a zero-length string is returned if the user doesn't enter anything in the text box prior to clicking **OK**.
5. *xpos* and *ypos* are specified in twips. A twip is a device-independent unit of measurement that equals 1/20 of a point or 1/1440 of an inch.
6. If the *xpos* parameter is omitted, the dialog box is centered horizontally.
7. If the *ypos* parameter is omitted, the top of the dialog box is positioned approximately one-third of the way down the screen.
8. If the *helpfile* parameter is provided, the *context* parameter must also be provided, and vice versa.
9. In **VBScript**, when both *helpfile* and *context* are passed to the **InputBox** function, a **Help** button is automatically placed on the **InputBox** dialog, allowing the user to click and obtain context-sensitive help.

Tips

- If you are omitting one or more optional arguments and using subsequent arguments, you must use a comma to signify the missing argument. For example, the following code fragment displays a prompt, a default string in the text box, and the help button, but default values are used for the title and positioning.

```
sString = InputBox("Enter it now", , "Something", , , "help.hlp", 321321)
```

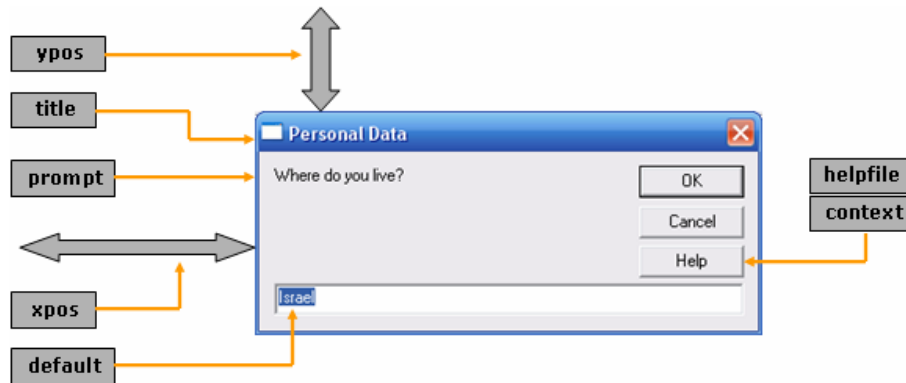


Figure 1 – InputBox function

Example

The following example uses the **InputBox** function to display an input box and assign the string to the variable Input:

```
Option Explicit
Dim sPrompt, sTitle, sHelpFile, sRes
Dim vDef
Dim nYPos, nXPos, nContext
sPrompt = "Where do you live?" : sTitle = "Personal Data"
vDef = "Israel"
nXPos = 100 : nYPos = 100 : nContext = 1001
sHelpFile = "C:\WINDOWS\system32\winhelp.hlp"
sRes = InputBox(sPrompt, sTitle, vDef, nXPos, nYPos, sHelpFile, nContext)
```

MsgBox Function

Description

Displays a dialog box containing a message, buttons, and optional icon to the user. The action taken by the user is returned by the function as an integer value.

Syntax

```
MsgBox (prompt, buttons, title, helpfile, context)
```

Arguments

Parameter	Description
<i>prompt</i>	The text of the message to display in the message box dialog.
<i>buttons</i>	The sum of the Button , Icon , Default Button , and Modality constant values.
<i>title</i>	The <i>title</i> displayed in the Title-bar of the message box dialog.
<i>helpfile</i>	An expression specifying the name of the help file to provide help functionality for the dialog.
<i>context</i>	An expression specifying a context ID within <i>helpfile</i> .

Notes

- *prompt* can contain approximately 1,000 characters, including carriage return characters such as the built-in **vbCrLf** or **vbNewLine** constants.
- In order to divide prompt onto multiple lines, you can use any of the **vbCr**, **vbLf**, **vbCrLf**, or **vbNewLine** constants. For example:

```
sMsg = "This is line 1" & vbCrLf & "This is line 2"
```

- If the *helpfile* parameter is provided, the *context* parameter must also be provided, and vice versa.
- When both *helpfile* and *context* are passed to the **MsgBox** function, a **Help** button is automatically placed on the **MsgBox** dialog, allowing the user to click and obtain context-sensitive help.
- If you omit the buttons argument, the default value is 0; **VBScript** opens an application modal dialog containing only an **OK** button.
- The following intrinsic constants can be added together to form a complete buttons argument:

```
ButtonDisplayConstant + IconDisplayConstant + _  
DefaultButtonConstant Or ModalityConstant
```

- Only one constant from each group can make up the overall *buttons* value.
- For a list of [Button Display Constants](#) see Table 2 **Error! Bookmark not defined..**
- Besides the message and the button(s), you can also display a friendly icon on the message box. To do that, combine the button value with one of the [Icons Display Constants](#)
- For a list of [Icons Display Constants](#) see Table 3 on page 128.
- When using the icon display constants, also the MS-Windows sound are affected.
- For a list of [Default Button Constants](#) see Table 4 on page 128.
- For a list of [Modality Constants](#) see Table 5 on page 128.
- For a list of [MsgBox Return Value Constants](#) see Table 1 on page 127.

Tips

- Application modality means that the user can't access other parts of the application until a response to the message box has been given. In other words, the appearance of the message box prevents the application from performing other tasks or from interacting with the user other than through the message box.
- System modality used to mean that all applications were suspended until a response to the message box was given. However, with multitasking operating systems such as the Windows family of 32- and 64-bit operating systems, this isn't the case. Basically the message box is defined to be a "Topmost" window that is set to "Stay on Top," which means that the user can switch to another application and use it without responding to the message box; but because the message box is the topmost window, it's positioned on top of all other running applications.
- Unlike its **InputBox** counterpart, **MsgBox** can't be positioned on the screen; it's always displayed in the center of the screen.

String Manipulation Functions

Asc, AscB, AsvW Functions

Description

The **Asc** function returns the ANSI (in the case of **Asc**) or Unicode (in the case of **AscW**) character code that represents the first character of the string passed to it. All other characters in the string are ignored. The **AscB** function returns the first byte of a string.

Syntax

```
Asc(string)
```

Arguments

Parameter	Description
<i>string</i>	The <i>string</i> argument is any valid string expression. If the <i>string</i> contains no characters, a run-time error occurs.

Return Value

An **integer** that represents the character code of the first character of the string.

Notes

- String expression passed to the function must contain at least one character, or a runtime error (either "Invalid use of Null" or "Invalid procedure call or argument") is generated.
- Only the first character of the string is evaluated by **Asc**, **AscB**, and **AscW**.
- Use the **AscW** function to return the Unicode character of the first character of a string.
- Use the **AscB** function to return the first byte of a string containing byte data.

Example

```
Dim sChars  
Dim nCharCode  
sChars = "QuickTest Professional"  
If Len(sChars) > 0 Then  
    nCharCode = Asc(sChars)  
    If nCharCode >= 97 And nCharCode <= 122 Then  
        MsgBox "The first character must be uppercase"  
    Else  
        MsgBox nCharCode  
    End If  
End If
```

Tips

- Always check that the string you are passing to the function contains at least one character using the **Len** function, as the following example shows:


```

If Len(sMyString) > 0 Then
    nCharCode = Asc(sMyString)
Else
    MsgBox "Cannot process a zero-length string"
End If

```

- Surprisingly, although the **VBScript** documentation shows that the data type of the parameter passed to the **Asc** function is String, it can actually be any data type.
- Evidently the **Asc** routine converts incoming values to strings before extracting their first character. Try this quick example for yourself:

```

sChars = 123
MsgBox Asc(sChars)

```

- Use **Asc** within your data validation routines to determine such conditions as whether the first character is upper- or lowercase and whether it's alphabetic or numeric, as the following example demonstrates:

```

Function CheckText (sText)
    Dim nChar
    If Len(sText) > 0 Then
        nChar = Asc(sText)
        If nChar >= 65 And nChar <= 90 Then
            CheckText = "The first character is UPPERCASE"
        ElseIf nChar >= 97 And nChar <= 122 Then
            CheckText = "The first character is lowercase"
        Else
            CheckText = "The first character isn't alphabetical"
        End If
    Else
        CheckText = "Please enter something in the text box"
    End If
End Function

```

Chr, ChrB, ChrW Functions

Description

Returns the character associated with the specified ANSI character code.

Syntax

```
Chr(charcode)
```

Arguments

Parameter	Description
<i>charcode</i>	The <i>charcode</i> argument is a number that identifies a character.

Return Value

Returns the character represented by *charcode*

Notes

- **Chr** returns the character associated with an ANSI character code.
- **ChrB** returns a one-byte string.
- **ChrW** returns a Unicode character.
- Numbers from 0 to 31 are the same as standard, nonprintable ASCII codes. For example, `Chr(10)` returns a linefeed character.

Tips

- Use **Chr**(34) to embed quotation marks inside a string, as shown in the following example:

```
sSQL = "SELECT * FROM tbl WHERE myCol=" & Chr(34) & sValue & Chr(34)
```

- You can use the **ChrB** function to assign binary values to String variables.
- The following table lists some of the more commonly used character codes that are supplied in the call to the **Chr** function:

Instr, InstrB Functions

Description

Returns the position of the first occurrence of one string within another.

Syntax

```
InStr (start, string1, string2, compare)
```

Arguments

Parameter	Description
<i>start</i>	Optional. Numeric expression that sets the starting position for each search. If omitted, search begins at the first character position. If <i>start</i> contains Null , an error occurs. The <i>start</i> argument is required if <i>compare</i> is specified.
<i>string1</i>	Required. String expression being searched.
<i>string2</i>	Required. String expression searched for.
<i>compare</i>	Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. See Settings section for values. If omitted, a binary comparison is performed.

Return Value

Variant of type Long.

Notes

- The return value of **InStr** is influenced by the values of *string1* (string to search) and *string2* (string to find)
- If the *start* argument is omitted, **InStr** commences the search with the first character of *string1*.
- If the *start* argument is **Null**, an error occurs.
- You must specify a *start* argument if you are specifying a *compare* argument.
- **VBScript** supports intrinsic constants for *compare*.
- For a list of [Comparison Constants](#) see Table 6 on page 128
- In effect, a binary comparison means that the search for *string2* in *string1* is case-sensitive. A text comparison means that the search for *string2* in *string1* is not case-sensitive.
- If the *compare* argument contains **Null**, an error is generated.
- If *compare* is omitted, the type of comparison is **vbBinaryCompare**.

Tips

You can use the **InStrB** function to compare byte data contained within a string. In this case, **InStrB** returns the byte position of *string2*, as opposed to the character position.

Example

```
nFind = InStr(4, "QuickTest Professional", "Test", 0)           ' returns 6
nFind = InStr(4, "QuickTest Professional", "test", 0)          ' returns 0
nFind = InStr(4, "QuickTest Professional", "test", 1)          ' returns 6
nFind = InStr( , "QuickTest Professional", "test")              ' returns 0
nFind = InStr("QuickTest Professional", "Test")                 ' returns 6
nFind = InStr("QuickTest Professional", "test", 1)              ' Type Mismatch
```

The following example demonstrates a possible usage of the **InStr** and **Chr** functions. The example retrieves all the instances of the searched string and returns the number of instances detected, also a reference array will give you additional information about the locations of the searched string.

```
Public Function InStrAll( _
    ByVal nStart, ByVal sString1, ByVal sString2, _
    ByVal nCompare, ByRef arrRes)

    Dim nBookMark, nFound, nCount
    Dim i
    nCount = 0 : nBookMark = nStart
    Do
        '--- searching the position found.
        nFound = InStr(nBookMark, sString1, sString2, nCompare)
        If nFound > 0 Then
            nCount = nCount + 1
            '--- resizing the array with preserve to store new location
            ReDim Preserve arrRes(nCount - 1)
            arrRes(nCount - 1) = nFound
            '--- setting the next start point to search
            nBookMark = nFound + 1
        End If
    Loop
```

```
Loop While nFound > 0
InStrAll = nCount
End Function
```

And here's a proposal code to activate the function:

```
Dim sText, sSearch
Dim arrStr()
sSearch = "to"
sText = "VBScript talks to host applications using Windows Script. " & _
        "With Windows Script, browsers and other host applications " & _
        "do not require special integration code for " & _
        "each scripting component. Windows Script enables a host " & _
        "to compile scripts, obtain and " & _
        "call entry points, and manage the namespace available to " & _
        "the developer. With Windows Script, language vendors " & _
        "can create standard language run times for scripting. " & _
        "Microsoft will provide run-time support for VBScript. " & _
        "Microsoft is working with various Internet groups to define " & _
        "the Windows Script standard so that scripting engines " & _
        "can be interchangeable. Windows Script is used in " & _
        "Microsoft® Internet Explorer and in Microsoft® " & _
        "Internet Information Service."
'--- calls the external function.
nInstancesCount = InStrAll(1, sText, sSearch, vbBinaryCompare, arrStr)
MsgBox Chr(34) & sSearch & Chr(34) & " was found " & _
        nInstancesCount & " times." & vbNewLine & _
        "The Positions found are: " & Join(arrStr, ",")
```

InStrRev Function

Description

The **InStrRev** function determines the starting position of a substring within a string by searching from the end of the string to its beginning.

Syntax

```
InStrRev (string1, string2, start, compare)
```

Arguments

Parameter	Description
<i>string1</i>	Required. String expression being searched.
<i>string2</i>	Required. String expression being searched for.
<i>start</i>	Required. String expression searched for.
<i>compare</i>	Optional. Numeric expression that sets the starting position for each search. If omitted, -1 is used, which means that the search begins at the last character position. If <i>start</i> contains Null , an error occurs.

Return Value

Variant of type Long.

Notes

- While **InStr** searches a string from left to right, **InStrRev** searches a string from right to left.
- **vbBinaryCompare** is case-sensitive; that is, **InStrRev** matches both character and case, whereas **vbTextCompare** is case-insensitive, matching only character, regardless of case.
- The default value for *compare* is **vbBinaryCompare**.
- *start* designates the starting point of the search and is the number of characters from the start of the string.
- If *start* is omitted, the search begins from the last character in *string1*.
- *string1* is the complete string in which you want to find the starting position of a substring.
- If *string2* isn't found, **InStrRev** returns 0.
- If *string2* is found within *string1*, the value returned by **InStrRev** is the position of *string1* from the start of the string.
- For a list of [Comparison Constants](#) see Table 6 on page 128

LCase, UCase Functions

Description

LCase function converts a string to lowercase.

UCase function converts a string to uppercase.

Syntax

```
LCase(string)
UCase(string)
```

Arguments

Parameter	Description
<i>string</i>	The <i>string</i> argument is any valid string expression. If <i>string</i> contains Null , Null is returned.

Return Value

Variant of type String.

Notes

- **LCase** affects only uppercase letters; all other characters in string are unaffected.
- **LCase** returns **Null** if string contains a **Null**.
- **UCase** affects only lowercase alphabetical letters; all other characters within string remain unaffected.
- **UCase** returns **Null** if string contains a **Null**.
- The **LCase** and **UCase** functions, helps the **QuickTest** programmer to avoid case-sensitive coding when reading data, from a data source i.e. the **DataTable** object

Example

The example, implements a **Select Case** branching, according to an output parameter.

```
Select Case sParam
Case "check_value"
Case "set_date"
:
:
End Select
```

Now, somehow in your script sParam got the value "Set_Date"
The select case will not work for sParam, so, to avoid the case-sensitive parameter use :

```
Select Case LCase(sParam)
Case "check_value"
Case "set_date"
:
:
End Select
```

Left, LeftB Functions

Description

The **Left** function returns a specified number of characters from the left side of a string.

Syntax

Left (*string*, *length*)

Arguments

Parameter	Description
<i>string</i>	String expression from which the leftmost characters are returned. If <i>string</i> contains Null , Null is returned.
<i>length</i>	Numeric expression indicating how many characters to return. If 0, a zero-length string ("") is returned. If greater than or equal to the

	number of characters in <i>string</i> , the entire string is returned.
--	--

Return Value

Variant of type String.

Notes

- If *length* is 0, a zero-length string (" ") is returned.
- If *length* is greater than the length of *string*, *string* is returned.
- If *length* is less than 0 or **Null**, the function generates runtime error
- If *string* contains **Null**, **Left** returns **Null**.
- **Left** processes strings of characters; **LeftB** is used to process binary data.

Tips

- Use the **Len** function to determine the overall length of string.
- When you use the **LeftB** function with byte data, length specifies the number of bytes to return.

Right, RightB Functions

Description

The **Right** function returns a specified number of characters from the right side of a string.

Syntax

Right (<i>string</i> , <i>length</i>)
--

Arguments

Parameter	Description
string	String expression from which the rightmost characters are returned. If <i>string</i> contains Null , Null is returned.
length	Numeric expression indicating how many characters to return. If 0, a zero-length string is returned. If greater than or equal to the number of characters in <i>string</i> , the entire string is returned.

Return Value

Variant of type String.

Notes

- If *length* is 0, a zero-length string (" ") is returned.
- If *length* is greater than the length of *string*, *string* is returned.
- If *length* is less than 0 or **Null**, the function generates a runtime error.
- If *string* contains **Null**, **Right** returns **Null**.
- **Right** processes strings of characters; **RightB** is used to process binary data.

Tips

- Use the **Len** function to determine the overall length of string.

- When you use the **RightB** function with byte data, length specifies the number of bytes to return.

Example

The following code assumes it's passed either a filename or a complete path and filename, and stores in a variable the filename from the end of the string:

```
strFullPath = "C:\Documents and Settings\My Documents\MySong.mp3"
lngStart = 1
Do
    lngPos = InStr(lngStart, strFullPath, "\")
    If lngPos = 0 Then
        strFilename = Right(strFullPath, Len(strFullPath) - lngStart + 1)
    Else
        lngStart = lngPos + 1
    End If
Loop While lngPos > 0
MsgBox strFilename
```

Mid, MidB Functions

Description

The **Mid** function returns a specified number of characters from a string. The **MidB** function is used with byte data contained in a string. Instead of specifying the number of characters, the arguments specify numbers of bytes.

Syntax

```
Mid(string, start, length)
```

Arguments

Parameter	Description
<i>string</i>	String expression from which characters are returned. If <i>string</i> contains Null , Null is returned.
<i>start</i>	Character position in <i>string</i> at which the part to be taken begins. If <i>start</i> is greater than the number of characters in <i>string</i> , Mid returns a zero-length string ("").
<i>length</i>	Number of characters to return. If omitted or if there are fewer than <i>length</i> characters in the text (including the character at <i>start</i>), all characters from the <i>start</i> position to the end of the string are returned.

Return Value

Returns a substring of a specified length from within a given string.

Notes

- If *string* contains a **Null**, **Mid** returns **Null**.
- If *start* is more than the length of *string*, a zero-length string is returned.
- If *start* is less than zero, the function generates a runtime error.
- If *length* is omitted or is greater than the *length* of *string*, all characters from *start* to the end of string are returned.

- The **MidB** version of the **Mid** function is used with byte data held within a string. When using **MidB**, both start and length refer to numbers of bytes as opposed to numbers of characters.

Tips

- Use the **Len** function to determine the overall length of string.
- Use **InStr** to determine the starting point of a given substring within another string.

Len, LenB Functions

Description

The **Len** function returns the number of characters in a string or the number of bytes required to store a variable.

The **LenB** function returns the number of bytes used to represent that string.

Syntax

```
Len(String | Varname)
```

Arguments

Parameter	Description
<i>String</i>	Any valid string expression. If <i>string</i> contains Null , Null is returned.
<i>Varname</i>	Any valid variable name. If <i>varname</i> contains Null , Null is returned.

Return Value

Variant of type Long.

Notes

- *string* and *varname* are mutually exclusive; that is, you must specify either string or *varname*, but not both.
- If either *string* or *varname* is **Null**, **Len** and **LenB** return **Null**.
- You can't use **Len** or **LenB** with an object variable.
- If *varname* is an array, you must also specify a valid subscript. In other words, **Len** and **LenB** can't determine the total number of elements in or the total size of an array. To determine the size of an array, use the **LBound** and **UBound** functions.

Tips

- Nonstring data is treated the same as strings when passed to the **Len** and **LenB** functions. For example, in the code:

```
Dim number, nLen
number = 1000
nLen = Len(number)
```

- The **Len** function returns 3, since that is the number of characters in the value of *number*.
- **LenB** is intended to work with string data, and returns the number of bytes required to store that string.

- If a non-string data type is passed to the **LenB** function, its value is first converted to a string before its length is determined.

LTrim, RTrim and Trim Functions

Description

Returns a copy of a string without leading spaces (**LTrim**), trailing spaces (**RTrim**), or both leading and trailing spaces (**Trim**).

Syntax

```
LTrim(string)  
RTrim(string)  
Trim(string)
```

Arguments

Parameter	Description
<i>string</i>	The <i>string</i> argument is any valid string expression. If <i>string</i> contains Null , Null is returned.

Return Value

Variant of type String.

Notes

- If *string* contains a **Null**, **LTrim** returns **Null**.
- If *string* contains a **Null**, **RTrim** returns **Null**.
- If *string* contains a **Null**, **Trim** returns **Null**.

Tips

- Unless you need to keep trailing spaces, it's best to use the **Trim** function, which is the equivalent of **LTrim(RTrim(string))**. This allows you to remove both leading and trailing spaces in a single function call.

StrComp Function

Description

The **StrComp** function returns a value indicating the result of a string comparison.

Syntax

```
StrComp(string1, string2, compare)
```

Arguments

Parameter	Description
<i>string1</i>	Required. Any valid string expression.
<i>string2</i>	Required. Any valid string expression.
<i>compare</i>	Optional. Numeric value indicating the kind of comparison to use when

	evaluating strings. If omitted, a binary comparison is performed. See Settings section for values.
--	--

Notes

- The compare argument can have one of the [Comparison Constants](#)
- For a list of [Comparison Constants](#) see Table 6 on page 128
- A comparison of two sequences of characters. Unless specified in the function making the comparison, all string comparisons are binary. In English, binary comparisons are case-sensitive; text comparisons are not.

Return Values

If	StrComp returns
<i>string1</i> is less than <i>string2</i>	-1
<i>string1</i> is equal to <i>string2</i>	0
<i>string1</i> is greater than <i>string2</i>	1
<i>string1</i> or <i>string2</i> is Null	Null

Example

The **StrComp** function is the most used in **QuickTest**. Always use this function when comparing with two strings. However, not always you need to know if the return value is -1 or 1, you only want to know if two string expressions are equal. You can use a general function for this matter as demonstrated below. We want the function to return **True** if *string1*=*string2*, so we convert the result to **Boolean** sub-type; but the **StrComp** returns 0 when two expressions are the same, so we need to perform a logical negation on the returned expression (**Not**).

```
Public Function CompStr(ByVal string1, ByVal string2, ByVal compare)
    If IsNull(string1) Or IsNull(string2) Then
        CompStr = Null
        Exit Function
    End If
    CompStr = Not CBool(StrComp(string1, string2, compare))
End Function
```

String Function

Description

The **String** function returns a repeating character string of the length specified.

Syntax

```
String(number, character)
```

Arguments

Parameter	Description
<i>number</i>	Length of the returned string. If <i>number</i> contains Null , Null is returned.
<i>character</i>	Character code specifying the character or string expression whose first character is used to build the return string. If <i>character</i> contains Null ,

	Null is returned.
--	--------------------------

Return Value

A string made up of character, repeated number times.

Notes

- If *number* contains **Null**, **Null** is returned.
- If *character* contains **Null**, **Null** is returned.
- If *character* consists of multiple characters, the first character is used only, and the remainders are discarded.
- If you specify a number for *character* greater than 255, **String** converts the number to a valid character code using the formula: *character* **Mod** 256

Tips

- The **String** function is useful for creating long strings of _, -, or = characters to create horizontal lines for delimiting sections of a report.

Replace Function

Description

The **Replace** function returns a string in which a specified substring has been replaced with another substring a specified number of times.

Syntax

```
Replace(expression, find, replacewith, start, count, compare)
```

Arguments

Parameter	Description
<i>expression</i>	Required. String expression containing substring to replace.
<i>find</i>	Required. Substring being searched for.
<i>replacewith</i>	Required. Replacement substring.
<i>start</i>	Optional. Position within <i>expression</i> where substring search is to begin. If omitted, 1 is assumed.
<i>count</i>	Optional. Number of substring substitutions to perform. If omitted, the default value is -1, which means make all possible substitutions.
<i>compare</i>	Optional. Numeric value indicating the kind of comparison to use when evaluating strings. If omitted, a binary comparison is performed. The <i>compare</i> argument can have one of the Comparison Constants values

Return Values

If	Replace returns
<i>expression</i> is zero-length	Zero-length string ("").
<i>expression</i> is Null	An error.
<i>find</i> is zero-length	Copy of <i>expression</i> .
<i>replacewith</i> is zero-length	Copy of <i>expression</i> with all occurrences of <i>find</i> removed.

<code>start > Len(expression)</code>	Zero-length string.
<code>count</code> is 0	Copy of <i>expression</i> .

Notes

- If *start* is omitted, the search begins at the start of the string.
- If *count* is omitted, its value defaults to -1, which means that all instances of the substring after *start* are replaced.
- **vbBinaryCompare** is case-sensitive; that is, Replace matches both character and case, whereas **vbTextCompare** is case-insensitive, matching only character, regardless of case.
- The default value for compare is **vbBinaryCompare**.
- *start* not only specifies where the search for *replacewith* begins, but also where the new string returned by the **Replace** function commences.
- The return value of the **Replace** function is a string, with substitutions made, that begins at the position specified by *start* and concludes at the end of the *expression* string. It is not a copy of the original string from *start* to finish.
- For a list of [Comparison Constants](#) see Table 6 on page 128

Tips

- If *count* isn't used, be careful when replacing short strings that may form parts of unrelated words. For example, consider the following:

```
Dim sString
sString = "You have to be careful when you do this, you could " & _
         "ruin your string"
MsgBox Replace(sString, "you", "we")
```

- Because we don't specify a value for *count*, the call to **Replace** replaces every occurrence of "you" in the original string with "we." But the fourth occurrence of "you" is part of the word "your," which is modified to become "wer."
- The best way to avoid this problem is to use regular expressions. By specifying the word-break pattern in your search criterion, you can insure that only whole words matched. For example:

```
sSearch = "You have to be careful when you do this " & _
         & "or you could ruin your string for you."
RegExp.Pattern = "you"
str = oRegExp.Replace(sSearch, "we")
MsgBox str
```

- You must also be aware that if *start* is greater than 1, the returned string starts at that character and not at the first character of the original string, as you might expect. For example, given the statements:

```
sOld = "This string checks the Replace function"
sNew = Replace(sOld, "check", "test", 5, vbTextCompare)
'--- sNew will contain the value: "string tests the Replace function"
```

Example

Suppose you want to save a text file in your system, based on the date and time. The **VBScript** function **Now()** returns

#4/24/2006 11:03:07 PM# (depend on Regional settings)

Is illegal, on Windows OS, to name a file with that syntax. First you have to convert the **Date** expression to a string, than remove the backslashes and colons signs (in other settings also dots and minus signs).

It seems a hard task, to use other functions and loops. The **Replace** function will fit this task:

```
Dim sNow, sFileName
'--- Converting to sub-type string
sNow = Cstr(Now())
'--- Replace backslashes with Null String
sFileName = Replace(sNow, "\", vbNullString, 1, -1, vbBinaryCompare)
'--- Replace colons with Null String
sFileName = Replace(sFileName, ":", vbNullString)
sFileName = sFileName & ".txt"
sFileName will contain the value: "4242006 110833 PM.txt"
```

StrReverse Function

Description

The **StrReverse** function returns a string that is the reverse of the string passed to it. For example, if the string "and" is passed to it as an argument, **StrReverse** returns the string "dna."

Syntax

```
StrReverse(stringexpression)
```

Arguments

Parameter	Description
<i>stringexpression</i>	Required. The string whose characters are to be reversed.

Return Value

A **String**.

Notes

- If *stringexpression* is a zero-length string (" "), the function's return value is a zero-length string.
- If *stringexpression* is **Null**, the function generates runtime error 94, "Invalid use of Null."

Space Function

Description

The **Space** function creates a string containing number spaces.

Syntax

```
Space(number)
```

Arguments

Parameter	Description
<i>number</i>	Required. An expression evaluating to the number of spaces required.

Return Value

A **String** containing number spaces.

Notes

- While *number* can be zero (in which case the function returns a empty string), runtime error 5, "Invalid procedure call or argument," is generated if *number* is negative.
- Space is a "convenience function" that is equivalent to the function call:

```
sString = String(number, 32)
```

DateTime Manipulation Functions

Date Function

Description

The **Abs** returns the current system date.

Syntax

```
CDate()
```

Return Value

Date returns a Date.

Tips

- To return both the current date and time in one variable, use the **Now** function.

DateAdd Function

Description

The function **DateAdd** returns a **Date** representing the result of adding or subtracting a given number of time periods to or from a given date or time. For instance, you can calculate the date 178 months before today's date, or the date and time 12,789 minutes from now.

Syntax

```
DateAdd(interval, number, date)
```

Arguments

Parameter	Description
<i>interval</i>	Required. An expression denoting the interval of time you need to add or

	subtract (see the following table "Interval Settings"). The <i>interval</i> argument can have on of the Interval Settings values.
<i>number</i>	Required. An expression denoting the number of time intervals you want to add or subtract.
<i>date</i>	Required. The date on which to base the DateAdd calculation.

Return Value

A Date Variant sub-type

Notes

- Specify the interval value as a string enclosed in quotation marks (e.g., "ww").
- If *number* is positive, the result will be after date; if *number* is negative, the result will be before date.
- The **DateAdd** function has a built-in calendar algorithm to prevent it from returning an invalid date. For example, if you add 10 minutes to 31 December 1999 23:55, **DateAdd** automatically recalculates all elements of the date to return a valid date; in this case, 1 January 2000 00:05. In addition, the calendar algorithm takes the presence of 29 February into account for leap years.
- For a list of [Interval Settings](#) see Table 7 on page 129

Tips

- When working with dates, always check that a date is valid using the **IsDate** function prior to passing it as a parameter to the function.
- To add a number of days to date, use either the day of the year "y", the day "d", or the weekday "w".
- The Variant date type can handle only dates as far back as 100 A.D. **DateAdd** generates an error, if the result precedes the year 100.
- The Variant date type can handle dates as far into the future as 9999 A.D.—from a practical application standpoint, a virtual infinity. If the result of DateAdd is a year beyond 9999 A.D., the function generates runtime error number 5.
- If number contains a fractional value, it's rounded to the nearest whole number before being used in the calculation.

DateDiff Function

Description

The **DateDiff** function calculates the number of time intervals between two dates. For example, you can use the function to determine how many days there are between 1 January 1980 and 31 May 1998.

Syntax

```
DateDiff(interval, date1, date2, firstdayofweek, firstweekofyear )
```

Arguments

Parameter	Description
-----------	-------------

<i>interval</i>	Required. The units of time used to express the result of the difference between <i>date1</i> and <i>date2</i> (see the following "Interval Settings" table). The <i>interval</i> argument can have one of the Interval Settings values.
<i>date1</i>	Required. The first date you want to use in the differential calculation.
<i>date2</i>	Required. The second date you want to use in the differential calculation.
<i>firstdayofweek</i>	Optional. A numeric constant that defines the first day of the week. If not specified, Sunday is assumed.
<i>firstdayofyear</i>	A numeric constant that defines the first week of the year. If not specified, the first week is assumed to be the week in which January 1 occurs

Return Value

A **Long** specifying the number of time intervals between two dates.

Notes

- The calculation performed by **DateDiff** is always between *date2* and *date1*. Therefore, if *date1* chronologically follows *date2*, the value returned by the function is negative.
- If interval is Weekday "w", **DateDiff** returns the number of weeks between *date1* and *date2*. **DateDiff** totals the occurrences of the day on which *date1* falls, up to and including *date2*, but not including *date1*. Note that an interval of "w" doesn't return the number of weekdays between two dates, as you might expect.
- If interval is Week "ww", **DateDiff** returns the number of calendar weeks between *date1* and *date2*. To achieve this, **DateDiff** counts the number of Sundays (or whichever other day is defined to be the first day of the week by the *firstdayofweek* argument) between *date1* and *date2*. *date2* is counted if it falls on a Sunday, but *date1* isn't counted, even if it falls on a Sunday.
- The *firstdayofweek* argument affects only calculations that use the "ww" (week) interval values.
- For a list of [Interval Settings](#) see Table 7 on page 129

Tips

- When working with dates, always check that a date is valid using the **IsDate** function prior to passing it as a function parameter.
- When comparing the number of years between December 31 of one year and January 1 of the following year, **DateDiff** returns 1, although in reality, the difference is only one day.
- **DateDiff** considers the four quarters of the year to be January 1-March 31, April 1-June 30, July 1-September 30, and October 1-December 31. Consequently, when determining the number of quarters between March 31 and April 1 of the same year, for example, **DateDiff** returns 1, even though the latter date is only one day after the former.
- If interval is "m", **DateDiff** simply counts the difference in the months on which the respective dates fall. For example, when determining the number of months between January 31 and February 1 of the same year, **DateDiff**

returns 1, even though the latter date is only one day after the former.

- To calculate the number of days between *date1* and *date2*, you can use either Day of year "y" or Day "d".
- In calculating the number of hours, minutes, or seconds between two dates, if an explicit time isn't specified, **DateDiff** provides a default value of midnight (00:00:00).
- If you specify *date1* or *date2* as strings within quotation marks (" ") and omit the year, the year is assumed to be the current year, as taken from the computer's date. This allows the same code to be used in different years.

DatePart Function

Description

The **DateDiff** function extracts an individual component of the date or time (like the month or the second) from a date/time value. It returns an Integer containing the specified portion of the given date. **DatePart** is a single function encapsulating the individual **Year**, **Month**, **Day**, **Hour**, **Minute**, and **Second** functions.

Syntax

```
DatePart(interval, date ,firstdayofweek , firstweekofyear )
```

Arguments

Parameter	Description
<i>interval</i>	Required. The unit of time to extract from within <i>date</i> (see the following table "Interval Settings"). The <i>interval</i> argument can have on of the Interval Settings values.
<i>date</i>	Required. The Date value that you want to evaluate.
<i>firstdayofweek</i>	Optional. A numeric constant that defines the first day of the week. If not specified, Sunday is assumed.
<i>firstdayofyear</i>	A numeric constant that defines the first week of the year. If not specified, the first week is assumed to be the week in which January 1 occurs

Return Value

An Integer.

Notes

- The *firstdayofweek* argument affects only calculations that use either the "w" or "ww" interval values.
- The *firstdayofweek* argument affects only calculations that use the "ww" interval value.
- For a list of [Interval Settings](#) see Table 7 on page 129

Tips

- When working with dates, always check that a date is valid using the **IsDate** function prior to passing it as a function parameter.
- If you specify date within quotation marks (" ") omitting the year, the year is assumed to be the current year taken from the computer's date.
- If you attempt to extract either the hours, the minutes, or the seconds, but date1 doesn't contain the necessary time element, the function assumes a time of midnight (0:00:00).

DateSerial Function

Description

The **DateSerial** function returns a **Date** from the three date components (year, month, and day). For the function to succeed, all three components must be present and all must be numeric values.

Syntax

```
DateSerial(year, month, day)
```

Arguments

Parameter	Description
<i>year</i>	Required. Number between 0 and 9999, inclusive.
<i>month</i>	Required. Any numeric expression to express the month (1-12).
<i>day</i>	Required. Any numeric expression to express the day (1-31).

Return Value

A Date.

Notes

- If the value of a particular element exceeds its normal limits, **DateSerial** adjusts the date accordingly. For example, if you tried **DateSerial** (96,2,31) - February 31, 1996 - **DateSerial** returns March 2, 1996.
- You can specify expressions or formulas that evaluate to individual date components as parameters to **DateSerial**. For example, **DateSerial** (98,10+9,23) returns 23 March 1999. This makes it easier to use **DateSerial** to form dates whose individual elements are unknown at design time or that are created on the fly because of user input.

Tips

- If any of the parameters exceed the range of the Integer data type (-32,768 to 32,767), an error (runtime error 6, "Overflow") is generated.
- The **Microsoft** documentation for this function incorrectly states, "For the year argument, values between 0 and 99, inclusive, are interpreted as the years 1900-1999." In fact, **DateSerial** handles two-digit years in the same way as other Visual Basic date functions. A year argument between 0 and 29 is taken to be in the 21st Century (2000 to 2029); year arguments between 30 and 99 are taken to be in the 20th Century (1930 to 1999). Of course, the safest way to specify a year is to use the full four digits.

DateValue Function

Description

The **DateValue** function returns a Date variant containing the date represented by *stringexpression*. **DateValue** can successfully recognize a *stringexpression* in any of the date formats recognized by **IsDate**. **DateValue** doesn't return time values in a date/time string; they are simply dropped. However, if *stringexpression* includes a valid date value but an invalid time value, a runtime error results.

Syntax

```
DateValue(stringexpression)
```

Arguments

Parameter	Description
<i>stringexpression</i>	Required. Number between 0 and 9999, inclusive.

Return Value

Variant of type Date.

Notes

- The order of the day, the month, and the year within *stringexpression* must be the same as the sequence defined by the computer's regional settings.
- Only those date separators recognized by **IsDate** can be used.
- If you don't specify a year in your date expression, **DateValue** uses the current year from the computer's system date.

Tips

- When working with dates, always check that a date is valid using the **IsDate** function prior to passing it as a function parameter.
- If *stringexpression* includes time information as well as date information, the time information is ignored; however, if only time information is passed to **DateValue**, an error is generated.
- **DateValue** handles two-digit years in the following manner: year arguments between 0 and 29 are taken to be in the 21st Century (2000 to 2029), and year arguments between 30 and 99 are taken to be in the 20th Century (1930 to 1999). The safest way to specify a year is to use the full four digits.
- The current formats being used for dates are easier to discover on Windows; the date formats are held as string values in the following registry keys:
 - Date Separator
HKEY_CURRENT_USER\Control Panel\International, sDate value entry
 - Long Date
HKEY_CURRENT_USER\Control Panel\International, sLongDate value entry
 - Short Date
HKEY_CURRENT_USER\Control Panel\International, sShortDate value entry

Day Function

Description

The **Day** function returns a variant integer data type that can take on a value ranging from 1 to 31, representing the day of the month of *dateexpression*. *dateexpression*, the argument passed to the **Day** function, must be a valid date/time or time value.

Syntax

```
Day (dateexpression)
```

Arguments

Parameter	Description
<i>dateexpression</i>	Required. Any expression capable of conversion to a Date.

Return Value

Variant of type Integer.

Notes

- *dateexpression* can be any variant, numeric expression, or string expression that represents a valid date.
- The range of *dateexpression* is 1/1/100 to 12/31/9999.
- If *dateexpression* is **Null**, **Null** is returned.

Tips

- When working with dates, always check that a date is valid using the **IsDate** function prior to passing it as a function parameter.
- If *dateexpression* omits the year, **Day** still returns a valid day.
- If the day portion of *dateexpression* is outside its valid range, the function generates runtime error 13, "Type mismatch." This is also true if the day and month portion of *dateexpression* is 2/29 for a non-leap year.
- To return the day of the week, use the **WeekDay** function.

Hour Function

Description

The **Hour** function extracts the hour element from a time expression.

Syntax

```
Hour (time)
```

Arguments

Parameter	Description
<i>time</i>	Required. Any valid time expression.

Return Value

A variant of data type Integer representing the hour of the day.

Notes

- **Hour** returns a whole number between 0 and 23, representing the hour of a 24-hour clock.
- If *time* contains **Null**, **Null** is returned.

Minute Function

Description

The **Hour** function returns an integer between 0 and 59 representing the minute of the hour from a given date/time expression.

Syntax

```
Minute(time)
```

Arguments

Parameter	Description
<i>time</i>	Required. Any valid date/time expression, or an expression that can be evaluated as a date/time expression.

Return Value

A variant of data type Integer representing the hour of the day.

Notes

- If *time* contains **Null**, **Null** is returned.

Tips

- If *time* isn't a valid date/time expression, the function generates runtime error 13, "Type mismatch." To prevent this, use the **IsDate** function to check the argument before calling the **Minute** function.

Month Function

Description

The **Month** function returns a variant representing the month of the year of a given date expression.

Syntax

```
Month(dateexpression)
```

Arguments

Parameter	Description
<i>dateexpression</i>	Required. Any expression capable of conversion to a Date.

Return Value

An Integer between 1 and 12.

Notes

- If *date* contains **Null**, **Month** returns **Null**.

Tips

- The validity of the date expression and the position of the month element within the date expression are initially determined by the locale settings of the current Windows system. However, some intelligence has been built into the **Month** function that surpasses the usual comparison of a date expression to the current locale settings. For example, on a Windows machine set to U.S. date format (mm/dd/yyyy), the date "13/12/2000" is technically illegal. However, the Month function returns 12 when passed this date.
- The basic rule for the **Month** function is that if the system-defined month element is outside legal bounds (i.e., greater than 12), the system-defined day element is assumed to be the month and is returned by the function.
- Since the **IsDate** function adheres to the same rules and assumptions as **Month**, it determines whether a date is valid before passing it to the **Month** function.

MonthName Function

Description

The **MonthName** function returns the month name of a given month. For example, 1 returns January, or if *abbreviate* is True, Jan.

Syntax

```
MonthName(monthnumber , abbreviate)
```

Arguments

Parameter	Description
<i>monthnumber</i>	Required. The ordinal number of the month, from 1 to 12.
<i>abbreviate</i>	Optional. A Boolean flag to indicate whether an abbreviated month name should be returned

Return Value

A String.

Notes

- The default value for *abbreviate* is **False**.

Tips

- *monthnumber* must be an integer or a long; it can't be a date. Use **DatePart**("m", dateval) to obtain a month number from a date.

Now Function

Description

The **Now** function returns the current date and time based on the system setting.

Syntax

```
Now ()
```

Return Value

A Date.

Tips

- If you convert the date returned by **Now** to a string, it takes the **Windows General Date format** based on the locale settings of the local computer. The U.S. setting for General Date is mm/dd/yy hh:mm:ss.
- The **Now** function is often used to generate timestamps. However, for short-term timing and intra-day timestamps, the **Timer** function, which returns the number of milliseconds elapsed since midnight, affords greater precision.

Example

The following example returns the date 10 days from today:

```
Dim dFutureDate  
dFutureDate = DateAdd("d", 10, Now)
```

Second Function

Description

The **Second** function extracts the seconds from a given time expression.

Syntax

```
Second(time)
```

Arguments

Parameter	Description
<i>time</i>	Required. Any valid expression that can represent a time value.

Return Value

An Integer in the range 0 to 59.

Notes

- If the *time* expression time is **Null**, the **Second** function returns **Null**.

Time Function

Description

The **Now** function returns the current system time.

Syntax

Time ()

Return Value

A Date.

Tips

- The **Time** function returns a value, but does not allow you to set the system time.

Timer Function

Description

The **Now** function returns the number of seconds since midnight..

Syntax

Timer ()

Return Value

A Single.

Tips

- You can use the **Timer** function as an easy method of passing a seed number to the **Randomize** statement, as follows:

```
Randomize Timer
```

Example

The following example measuring the time taken to execute a procedure

```
Option Explicit
Dim sStartTime
Dim nTime
sStartTime = Timer()
Wait 10
nTime = FormatDateTime(Timer - sStartTime, vbShortTime)
MsgBox nTime
```

TimeSerial Function

Description

The **TimeSerial** function Constructs a valid time given a number of hours, minutes, and seconds.

Syntax

TimeSerial(*hour, minute, second*)

Arguments

Parameter	Description
-----------	-------------

<i>hour</i>	Required. A number in the range 0 to 23.
<i>minute</i>	Required. Any numeric expression.
<i>second</i>	Required. Any numeric expression.

Return Value

A Date.

Notes

- Any of the arguments can be specified as relative values or expressions.
- The hour argument requires a 24-hour clock format; however, the return value is always in a 12-hour clock format suffixed with A.M. or P.M.
- If any of the values are greater than the normal range for the time unit to which it relates, the next higher time unit is increased accordingly. For example, a second argument of 125 evaluated as 2 minutes 5 seconds.
- If any of the values are less than zero, the next higher time unit is decreased accordingly. For example, **TimeSerial**(2,-1,30) returns 01:59:30.
- If any of the values are outside the range -32,768 to 32,767, an error occurs.
- If the value of any parameter causes the date returned by the function to fall outside the range of valid dates, an error occurs.

Tips

Because **TimeSerial** handles time units outside of their normal limits, it can be used for time calculations. However, because the **DateAdd** function is more flexible and is internationally aware, it should be used instead.

TimeValue Function

Description

The **TimeSerial** function converts a string representation of a time to a Variant Date type.

Syntax

```
TimeValue (time)
```

Arguments

Parameter	Description
<i>time</i>	Required. A number in the range 0 to 23.

Return Value

A Date.

Notes

- If time is invalid, a runtime error is generated.
- If time is **Null**, **TimeValue** returns **Null**.
- Both 12- and 24-hour clock formats are valid.

- Any date information contained within time is ignored by the **TimeValue** function.
- If **TimeValue** returns invalid time information, an error occurs.

Tips

A time literal can also be assigned to a **Variant** or **Date** variable by surrounding the date with hash characters (#), as the following snippet demonstrates:

```
Dim dMyTime
dMyTime = #12:30:00 AM#
```

The **CDate** function can also cast a time expression contained within a string as a Date variable, with the additional advantage of being internationally aware.

WeekDay Function

Description

The **WeekDay** function determines the day of the week of a given date.

Syntax

```
Weekday (date, [firstdayofweek])
```

Arguments

Parameter	Description
<i>Date</i>	Required. Any valid date expression.
<i>firstdayweek</i>	Optional. Integer specifying the first day of the week.

Return Value

The Weekday function can return any of the [WeekDay Costant](#) Values. For a list of [WeekDay Costant](#) see Table 14 on page 131

Tips

- If you specify a firstdayofweek argument, the function returns the day of the week relative to firstdayofweek. For instance, if you set the value of firstdayofweek to vbMonday (2), indicating that Monday is the first day of the week, and attempt to determine the day of the week on which October 1, 1996, falls, the function returns a 2. That's because October 1, 1996, is a Tuesday, the second day of a week whose first day is Monday.
- Because the function's return value is relative to firstdayofweek, using the day of the week constants to interpret the function's return value is confusing, to say the least. If we use our October 1, 1996, example once again, the following expression evaluates to True if the day of the week is Tuesday:

```
If vbMonday = WeekDay (CDate ("10/1/96"), vbMonday) Then
```

WeekDayName Function

Description

The **WeekDayName** function returns the real name of the day.

Syntax

```
WeekDayName (weekday, abbreviate , firstdayofweek )
```

Arguments

Parameter	Description
<i>weekday</i>	Required. The ordinal number of the required weekday, from 1 to 7.
<i>abbreviate</i>	Optional. Specifies whether to return the full day name or an abbreviation.
<i>firstdayofweek</i>	Optional. Specifies which day of the week should be first.

Return Value

A String.

Notes

- *weekday* must be a number between 1 and 7, or the function generates runtime error 5.
- If *firstdayofweek* is omitted, **WeekDayName** treats Sunday as the first day of the week.
- The default value of *abbreviate* is **False**.

Tips

You'd expect that, given a date, *weekday* would return the name of that date's day. But this isn't how the function works. To determine the name of the day of a particular date, combine **WeekDayName** with a call to the **WeekDay** function, as the following code fragment shows:

```
sDay = WeekDayName (WeekDay (dDate, nFirstDay), bFullName, nFirstDay)
```

Note that the value of the **FirstDayOfWeek** argument must be the same in the calls to both functions for **WeekDayName** to return an accurate result.

Year Function

Description

The **Year** function returns a whole number representing the year.

Syntax

```
Year (dateexpression)
```

Arguments

Parameter	Description
<i>dateexpression</i>	Required. Any expression capable of conversion to a Date.

Return Value

Returns an integer representing the year in a given date expression.

Notes

- If *date* contains **Null**, **Year** returns **Null**.

Tips

- The validity of the date expression and position of the year element within the given date expression are initially determined by the locale settings of the Windows system. However, some extra intelligence relating to two-digit year values has been built into the Year function that surpasses the usual comparison of a date expression to the current locale settings.
- What happens when you pass a date over to the **Year** function containing a two-digit year? Quite simply, when the **Year** function sees a two-digit year, it assumes that all values equal to or greater than 30 are in the 20th Century (i.e., 30 = 1930, 98 = 1998) and that all values less than 30 are in the 21st century (i.e., 29 = 2029, 5 = 2005). Of course, if you don't want sleepless nights rewriting your programs in the year 2029, you should insist on a four-digit year, which will see your code work perfectly for about the next 8,000 years!

Math Functions

Abs Function

Description

The **Abs** returns the absolute value of a number.

Syntax

```
Abs (number)
```

Arguments

Parameter	Description
<i>number</i>	The <i>number</i> argument can be any valid numeric expression. If <i>number</i> contains Null , Null is returned; if it is an un-initialized variable, zero is returned.

Return Value

The absolute value of the argument *number*. The data type is the same as that passed to the function if *number* is numeric, and **Double** if it is not.

Notes

- *number* can be a number, a string representation of a number, an object whose default property is numeric, or a **Null** or **Empty**.
- If *number* is **Null**, the function returns **Null**.
- If *number* is an uninitialized variable or **Empty**, the function returns zero.

Atn Function

Description

Atn function returns the arctangent of a number.

Syntax

```
Atn (number)
```

Arguments

Parameter	Description
<i>number</i>	Any numeric expression, representing the ratio of two sides of a right angle triangle.

Return Value

Atn function returns value is a Double representing the arctangent of *number* in the range -pi/2 to pi/2 radians.

Notes

- If no number specified, a runtime error is generated.
- The return value of **Atn** is in radians, not degrees.

Tips

- To convert degrees to radians, multiply degrees by pi/180.
- To convert radians to degrees, multiply radians by 180/pi.
- Do not confuse **Atn** with the cotangent. **Atn** is the inverse trigonometric function of **Tan**, as opposed to the simple inverse of **Tan**.

Example

The following example uses **Atn** to calculate the value of pi:

```
Private Const Pi = 3.14159
Dim fSideAdj, fSideOpp, fRatio, fAtangent, fDegrees
fSideAdj = 50.25 : fSideOpp = 75.5
fRatio = fSideOpp / fSideAdj
fAtangent = Atn(fRatio)
'-- Convert from radians to degrees
fDegrees = fAtangent * (180 / Pi)
MsgBox fDegrees & " Degrees"
```

Cos Function

Description

Takes an angle specified in radians and returns a ratio representing the length of the side adjacent to the angle divided by the length of the hypotenuse.

Syntax

```
Cos (number)
```

Arguments

Parameter	Description
<i>number</i>	The number argument can be any valid numeric expression that expresses an angle in radians.

Return Value

A type of **Double** denoting the cosine of an angle.

Notes

- If no number specified, a runtime error is generated.
- The cosine returned by the function is between -1 and 1.

Tips

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

Sin Function

Description

The **Sin** function returns the sine of an angle.

Syntax

```
Sin(number)
```

Arguments

Parameter	Description
<i>number</i>	The number argument can be any valid numeric expression that expresses an angle in radians.

Return Value

A **Double** containing the sine of an angle.

Notes

- If no number specified, a runtime error is generated.
- The ratio is determined by dividing the length of the side opposite the angle by the length of the hypotenuse.
- The result lies in the range -1 to 1.

Tips

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

Tan Function

Description

The **Tan** function returns the tangent of an angle.

Syntax

```
Tan(number)
```

Arguments

Parameter	Description
<i>number</i>	The number argument can be any valid numeric expression that expresses an angle in radians.

Return Value

A **Double** containing the tangent of an angle.

Notes

- If no number specified, a runtime error is generated.
- The returned ratio is derived by dividing the length of the side opposite the angle by the length of the side adjacent to the angle.

Tips

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

Log Function

Description

The **Log** function returns the natural logarithm of a number.

Syntax

```
Log(number)
```

Arguments

Parameter	Description
<i>number</i>	<i>number</i> argument can be any valid numeric expression greater than 0.

Return Value

A **Double** containing the Natural Logarithm of a number.

Notes

- If no number specified, a runtime error is generated.
- The natural logarithm is based on e , a constant whose value is approximately 2.718282. The natural logarithm is expressed by the equation: $e^z = x$ where $z = \text{Log}(x)$. In other words, the natural logarithm is the inverse of the exponential function.
- *number*, the value whose natural logarithm the function is to return, must be a positive real number. If number is negative or zero, the function generates a runtime error.

Tips

- You can calculate base-n logarithms for any number x, by dividing the natural logarithm of x by the natural logarithm of n, as the following expression illustrates:

```
Logn(x) = Log(x) / Log(n)
```

- The following example illustrates a custom Function that calculates base-10 logarithms:

```
Function Log10(X)
    Log10 = Log(X) / Log(10)
End Function
```

Exp Function

Description

The **Exp** function returns the antilogarithm of a number; the antilogarithm is the base of natural logarithms, *e* (whose value is the constant 2.7182818), raised to a power.

Syntax

```
Exp(number)
```

Arguments

Parameter	Description
<i>number</i>	The number argument can be any valid numeric expression.

Return Value

A **Double** representing the antilogarithm of *number*.

Notes

- If no number specified, a runtime error is generated.
- If the value of *number* exceeds 709.782712893, an error occurs
- The constant *e* is approximately 2.718282.

Tips

- **Exp** is the inverse of the **Log** function.

Sqr Function

Description

The **Sqr** function calculates the square root of a given number.

Syntax

```
Sqr(number)
```

Arguments

Parameter	Description
<i>number</i>	The number argument can be any valid numeric expression greater than or equal to 0.

Return Value

A **Double** containing the square root of *number*.

Notes

- If no number specified, a runtime error is generated.
- *number* must be equal to or greater than zero or a runtime error occurs.

Derived Math Functions

Description

A number of other mathematical functions that aren't intrinsic to VBScript can be computed using the value returned by the Log function. The functions and their formulas are:

Derived Math functions

Function	Derived equivalents
Secant	$\text{Sec}(X) = 1 / \text{Cos}(X)$
Cosecant	$\text{Cosec}(X) = 1 / \text{Sin}(X)$
Cotangent	$\text{Cotan}(X) = 1 / \text{Tan}(X)$
Inverse Sine	$\text{Arcsin}(X) = \text{Atn}(X / \text{Sqr}(-X * X + 1))$
Inverse Cosine	$\text{Arccos}(X) = \text{Atn}(-X / \text{Sqr}(-X * X + 1)) + 2 * \text{Atn}(1)$
Inverse Secant	$\text{Arcsec}(X) = \text{Atn}(X / \text{Sqr}(X * X - 1)) + \text{Sgn}((X) - 1) * (2 * \text{Atn}(1))$
Inverse Cosecant	$\text{Arccosec}(X) = \text{Atn}(X / \text{Sqr}(X * X - 1)) + (\text{Sgn}(X) - 1) * (2 * \text{Atn}(1))$
Inverse Cotangent	$\text{Arccotan}(X) = \text{Atn}(X) + 2 * \text{Atn}(1)$
Hyp Sine	$\text{HSin}(X) = (\text{Exp}(X) - \text{Exp}(-X)) / 2$
Hyp Cosine	$\text{HCos}(X) = (\text{Exp}(X) + \text{Exp}(-X)) / 2$
Hyp Tangent	$\text{HTan}(X) = (\text{Exp}(X) - \text{Exp}(-X)) / (\text{Exp}(X) + \text{Exp}(-X))$
Hyp Secant	$\text{HSec}(X) = 2 / (\text{Exp}(X) + \text{Exp}(-X))$
Hyp Cosecant	$\text{HCosec}(X) = 2 / (\text{Exp}(X) - \text{Exp}(-X))$
Hyp Cotangent	$\text{HCotan}(X) = (\text{Exp}(X) + \text{Exp}(-X)) / (\text{Exp}(X) - \text{Exp}(-X))$
Inverse Hyp Sine	$\text{HArcsin}(X) = \text{Log}(X + \text{Sqr}(X * X + 1))$
Inverse Hyp Cosine	$\text{HArccos}(X) = \text{Log}(X + \text{Sqr}(X * X - 1))$
Inverse Hyp Tangent	$\text{HArctan}(X) = \text{Log}((1 + X) / (1 - X)) / 2$
Inverse Hyp Secant	$\text{HArcsec}(X) = \text{Log}((\text{Sqr}(-X * X + 1) + 1) / X)$
Inverse Hyp Cosecant	$\text{HArccosec}(X) = \text{Log}((\text{Sgn}(X) * \text{Sqr}(X * X + 1) + 1) / X)$
Inverse Hyp Cotangent	$\text{HArccotan}(X) = \text{Log}((X + 1) / (X - 1)) / 2$
Logarithm to base N	$\text{LogN}(X) = \text{Log}(X) / \text{Log}(N)$

Int Function

Description

The **Int** function returns the integer portion of a number.

Syntax

```
Int(number)
```

Arguments

Parameter	Description
<i>number</i>	Any valid numeric data type, the number to be truncated.

Return Value

Returns a value of the numeric data type passed to it.

Notes

- The fractional part of *number* is removed and the resulting integer value returned. **Int** doesn't round number to the nearest whole number; for example, **Int**(100.9) returns 100.
- The difference between **Int** and **Fix** is that if number is negative, **Int** returns the first negative integer less than or equal to number, whereas **Fix** returns the first negative integer greater than or equal to number. For example, **Int** converts -8.4 to -9, and **Fix** converts -8.4 to -8.

Tips

- **Int** and **Fix** work identically with positive numbers. However, for negative numbers, **Fix** returns the first negative integer greater than *number*. For example, **Int**(-10.1) returns -10.
- Do not confuse the **Int** function with **CInt**. **CInt** casts the number passed to it as an Integer data type, whereas **Int** returns the same data type that was passed to it.

Example

The following examples illustrate how the **Int** function return integer portions of numbers

```
nInt = Int(99.8)      ' Returns 99.
nInt = Int(-99.8)     ' Returns -100.
nInt = Int(-99.2)     ' Returns -100.
nInt = Int(6.83227)   ' Returns 6
nInt = Int(6.23443)   ' Returns 6
nInt = Int(-6.13443)  ' Returns -7
nInt = Int(-6.93443)  ' Returns -7
```

Fix Function

Description

The **Fix** function returns the integer part of a specified number.

Syntax

```
Fix(number)
```

Arguments

Parameter	Description
<i>number</i>	Required. A valid numeric expression.

Return Value

The same data type as passed to the function containing only the integer portion of number.

Notes

- If number is Null, Fix returns **Null**.
- The operations of **Int** and **Fix** are identical when dealing with positive numbers: numbers are rounded down to the next lowest whole number. For example, both **Int**(3.14) and **Fix**(3.14) return 3.
- If number is negative, **Fix** removes its fractional part, thereby returning the next greater whole number. For example, **Fix**(-3.667) returns -3. This contrasts with **Int**, which returns the negative integer less than or equal to number (or -4, in the case of our example).

Tips

- **Fix** does not round number to the nearest whole number; it simply removes the fractional part of number. Therefore, the integer returned by **Fix** is the nearest whole number less than (or greater than, if the number is negative) the number passed to the function

Example

The following examples illustrate how the **Fix** function return integer portions of numbers

```
nFix = Fix(99.2)      ' Returns 99.  
nFix = Fix(-99.8)     ' Returns -99.  
nFix = Fix(-99.2)     ' Returns -99.  
nFix = Fix(6.83227)   ' Returns 6  
nFix = Fix(6.23443)   ' Returns 6  
nFix = Fix(-6.13443)  ' Returns -6  
nFix = Fix(-6.93443)  ' Returns -6
```

Hex Function

Description

The **Hex** function returns a string that represents the hexadecimal value of a specified number.

Syntax

```
Hex(number)
```

Arguments

Parameter	Description
<i>number</i>	The number argument is any valid expression.

Return Values

If <i>number</i> is	Hex returns
Null	Null
Empty	Zero (0)
Any other number	Up to eight hexadecimal characters.

Notes

- If *number* contains a fractional part, it's rounded automatically to the nearest whole number prior to processing. If the number ends in .5, it's rounded to the nearest even whole number.
- *number* must evaluate to a numeric expression that ranges from -2,147,483,648 to 2,147,483,647. If the argument is outside this range, runtime error 6, "Overflow," results.

Tips

You can represent hexadecimal numbers directly by preceding numbers in the proper range with &H. For example, &H10 represents decimal 16 in hexadecimal notation.

Example

The following example uses the **Hex** function to return the hexadecimal value of a number:

```
nDecimal = Hex(3)      ' Returns 3.
nDecimal = Hex(5)      ' Returns 5.
nDecimal = Hex(9)      ' Returns 9.
nDecimal = Hex(10)     ' Returns A
nDecimal = Hex(11)     ' Returns B
nDecimal = Hex(12)     ' Returns C
nDecimal = Hex(400)    ' Returns 190
nDecimal = Hex(459)    ' Returns 1CB
nDecimal = Hex(460)    ' Returns 1CC
```

Oct Function

Description

The **Hex** function returns a string representing the octal value of a number.

Syntax

```
Oct(number)
```

Arguments

Parameter	Description
<i>number</i>	Number or string representation of a number to convert.

Return Values

If number is	Hex returns
Null	Null
Empty	Zero (0)
Any other number	Up to 11 octal characters

Notes

- If number isn't already a whole number, it's rounded to the nearest whole number before being evaluated.

Tips

You can also use literals in your code to represent octal numbers by appending &O to the relevant octal value. For example, 100 decimal has the octal representation &O144.

Example

The following example uses the **Oct** function to return the octal value of a number

```
nDecimal = Oct(3)      ' Returns 3
nDecimal = Oct(5)      ' Returns 5
nDecimal = Oct(9)      ' Returns 11
nDecimal = Oct(10)     ' Returns 12
nDecimal = Oct(11)     ' Returns 13
nDecimal = Oct(12)     ' Returns 14
nDecimal = Oct(400)    ' Returns 620
nDecimal = Oct(459)    ' Returns 713
nDecimal = Oct(460)    ' Returns 714
```

Sgn Function

Description

The **Sgn** function returns an integer indicating the sign of a number.

Syntax

Sgn (*number*)

Arguments

Parameter	Description
<i>number</i>	The <i>number</i> argument can be any valid numeric expression.

Return Values

If number is	Hex returns
Null	Null
Empty	Zero (0)
Any other number	Up to 11 octal characters

Notes

- If number is not already a whole number, it is rounded to the nearest whole number before being evaluated.
- The sign of the number argument determines the return value of the **Sgn** function.

Tips

- If you're planning to use the **Sgn** function to evaluate a result to **False** (0) or **True** (any nonzero value), you could also use the **CBool** function.
- The major use for **Sgn** is to determine the sign of an expression.
- **Sgn** is useful in cases in which the sign of a quantity defines the sign of an expression.
- Although **Sgn** handles the conversion of strings to numeric data, it's a good idea to make sure that number is valid by calling the **IsNumeric** function before the call to **Sgn**.

Example

The example uses the **Sgn** function to determine the sign of a number:

```
nSign = Sgn(15)      ' Returns 3.
nSign = Sgn(-5.6)    ' Returns 5.
nSign = Sgn(0)       ' Returns 9.
```

Round Function

Description

The **Round** function rounds a given number to a specified number of decimal places.

Syntax

```
Round(expression, numdecimalplaces)
```

Arguments

Parameter	Description
<i>expression</i>	Required. Numeric expression being rounded.
<i>numdecimalplaces</i>	Optional. Number indicating how many places to the right of the decimal are included in the rounding. If omitted, integers are returned by the Round function.

Return Value

The same data type as expression.

Notes

- *numdecimalplaces* can be any whole number between 0 and 16.
- **Round** follows standard rules for rounding:
 - If the digit in the position to the right of *numdecimalplaces* is greater than 5, the digit in the *numdecimalplaces* position is incremented by 1.
 - If the digit in the position to the right of *numdecimalplaces* is less than

5, the digits to the right of *numdecimalplaces* are dropped.

- If the digit in the position to the right of *numdecimalplaces* is 5 and the digit in the *numdecimalplaces* position is odd, the digit in the *numdecimalplaces* position is incremented by 1.
- If the digit in the position to the right of *numdecimalplaces* is 5 and the digit in the *numdecimalplaces* position is even, the digits to the right of *numdecimalplaces* are dropped.

Tips

- If expression is a string representation of a numeric value, **Round** converts it to a numeric value before rounding. However, if *expression* isn't a string representation of a number, **Round** generates runtime **error 13**, "Type mismatch."
- The **IsNumeric** function insures that expression is a proper numeric representation before calling **Round**.

Example

The following example uses the **Round** function to round a number to some decimal places:

```
nRound = Round(3.14159)      ' Returns 3
nRound = Round(3.14159, 2)   ' Returns 3.14
nRound = Round(3.14159, 1)   ' Returns 3.1
```

Rnd Function

Description

The **Rnd** function returns a random number. The number is always less than 1 but greater or equal to 0.

Remarks

- Instead of use **Randomize** and **Rnd** functions, **QuickTest** supplies a function utility the **RandomNumber** object
- `RandomNumber(ParameterNameOrStartNumber [,EndNumber])`

Format Functions

FormatCurrency, FormatNumber and FormatPercent Functions

Description

- The **FormatCurrency** function returns an expression formatted as a currency value using the currency symbol defined in the computer's control panel.
- The **FormatNumber** function returns an expression formatted as a number.
- The **FormatPercent** function returns an expression formatted as a percentage (multiplied by 100) with a trailing % character.

Syntax

```
FormatCurrency (Expression[, NumDigitsAfterDecimal][, _  
    IncludeLeadingDigit[, UseParensForNegativeNumbers[, GroupDigits]]])  
FormatNumber (Expression[, NumDigitsAfterDecimal][, _  
    IncludeLeadingDigit[, UseParensForNegativeNumbers[, GroupDigits]]])  
FormatPercent (Expression[, NumDigitsAfterDecimal][, _  
    IncludeLeadingDigit[, UseParensForNegativeNumbers[, GroupDigits]]])
```

Arguments

Parameter	Description
<i>expression</i>	Required. Expression to be formatted.
<i>NumDigitsAfterDecimal</i>	Optional. Numeric value indicating how many places to the right of the decimal are displayed. Default value is -1, which indicates that the computer's regional settings are used.
<i>IncludeLeadingDigit</i>	Optional. Tristate constant that indicates whether or not a leading zero is displayed for fractional values. Indicates whether the formatted string is to have a 0 before floating-point numbers between 1 and -1. See Settings section for values.
<i>UseParensForNegativeNumbers</i>	Optional. Tristate constant that indicates whether or not to place negative values within parentheses. See Settings section for values.
<i>GroupDigits</i>	Optional. Tristate constant that indicates whether or not numbers are grouped using the group delimiter specified in the computer's regional settings. See Settings section for values.

Remarks

- When one or more optional arguments are omitted, values for omitted arguments are provided by the computer's regional settings.
- The position of the currency symbol relative to the currency value is determined by the system's regional settings.
- All settings information comes from the **Regional Settings Currency** tab, except leading zero which comes from the Number tab.
- The *IncludeLeadingDigit*, *UseParensForNegativeNumbers*, and *GroupDigits* arguments have one of [Tristate Constants](#) values
- For a list of [Tristate Constants](#) see Table 8 on page 129

Example

The following example uses the **FormatCurrency** function to format the expression as a currency and assign it to cCurr:

```
cCurr = FormatCurrency(1000.578)           ' Returns $1,000.58  
cCurr = FormatCurrency(1000.578, 1)       ' Returns $1,000.6  
cCurr = FormatCurrency(1000.578, 3)       ' Returns $1,000.578  
cCurr = FormatCurrency(.578,,True)        ' Returns $0.58  
cCurr = FormatCurrency(.578,,False)       ' Returns $.58  
cCurr = FormatCurrency(-1000.578,,,True)  ' Returns ($1,000.58)  
cCurr = FormatCurrency(-1000.578,,,False) ' Returns -$1,000.58  
cCurr = FormatCurrency(1000000.57,,,,True) ' Returns $1,000,000.57
```

```

cCurr = FormatCurrency(1000000.57,,,False) ' Returns $1000000.57
nNum = FormatNumber(1000.578) ' Returns 1,000.58
nNum = FormatNumber(1000.578, 1) ' Returns 1,000.6
nNum = FormatNumber(1000.578, 3) ' Returns 1,000.578
nNum = FormatNumber(.578,,True) ' Returns 0.58
nNum = FormatNumber(.578,,False) ' Returns $.58
nNum = FormatNumber(-1000.578,,,True) ' Returns (1,000.58)
nNum = FormatNumber(-1000.578,,,False) ' Returns -1,000.58
nNum = FormatNumber(1000000.578,,,,True) ' Returns 1,000,000.58
nNum = FormatNumber(1000000.578,,,,False) ' Returns 1000000.58
nPerc = FormatPercent(1000.578) ' Returns 57.80%
nPerc = FormatPercent(1000.578, 1) ' Returns 57.8%
nPerc = FormatPercent(1000.578, 3) ' Returns 57.800%
nPerc = FormatPercent(.578,,True) ' Returns 57.80%
nPerc = FormatPercent(.578,,False) ' Returns 57.80%
nPerc = FormatPercent(-1000.578,,,True) ' Returns (57.80%)
nPerc = FormatPercent(-1000.578,,,False) ' Returns -57.80%
nPerc = FormatPercent(1000000.578,,,,True) ' Returns 57.80%
nPerc = FormatPercent(1000000.578,,,,False) ' Returns 57.80%

```

FormatDateTime Function

Description

The **FormatDateTime** function returns an expression formatted as a date or time.

Syntax

```
FormatDateTime (Date[,NameFormat])
```

Arguments

Parameter	Description
<i>Date</i>	Required. Date expression to be formatted.
<i>NameFormat</i>	Optional. Numeric value that indicates the date/time format used. If omitted, vbGeneralDate is used.

Remarks

- The *NamedFormat* argument can be one of [Date Format Constants](#) value
- For a list of [Date Format Constants](#) see Table 10 on page 129
- **vbLongDate** Uses the long date format specified in the client computer's regional settings. For example:

```
MsgBox FormatDate Time(#04/10/03#, vbLongDate)
```

- displays Thursday, April 10, 2003.

- **vbShortDate** Uses the short date format specified in the client computer's regional settings. For example:

```
MsgBox FormatDate Time(#04/10/03#, vbShortDate)
```

- displays 4/102003.
- **vbLongTime** Uses the time format specified in the computer's regional settings. For example:

```
MsgBox FormatDate Time(#1:03:00 PM#, vbLongTime)
```

- displays 1:03:00 PM.
- **VbShortTime** Uses a 24-hour format (hh:mm). For example:

```
MsgBox FormatDate Time(#1:03:00 PM#, vbShortTime)
```

- displays 13:03.
- The default date format is **vbGeneralDate(0)**.
- These constants are all defined in the **VBScript** library and hence are an intrinsic part of the language.

Tips

- Remember that date and time formats obtained from the client computer are based on the client computer's regional settings.
- It's not uncommon for a single application to be used internationally, so that date formats can vary widely. Not only that, but you can never be sure that a user has not modified the regional settings on a computer.
- In short, never take a date coming in from a client machine for granted; ideally, you should always insure it's in the format you need prior to using it.

Data Type Validation Functions

VarType Function

Description

The **VarType** function returns a value indicating the subtype of a variable.

Syntax

```
VarType (varname)
```

Arguments

Parameter	Description
<i>varname</i>	The <i>varname</i> argument can be any variable.

Return Value

An Integer representing the data type of *varname*.

Notes

- One of the [VarType Constants](#) can test the return value of the **VarType** function:
- These constants are specified by **VBScript**. As a result, the names can be used anywhere in your code in place of the actual values
- If *varname* is an array created by **VBScript** code, the **VarType** function returns 8200 (**vbArray**) and **vbVariant**.
- If *varname* is an array returned to the script by a component, the **VarType** function returns 8200 (**vbArray**) and the value representing the data type of the array. For instance, a **Visual Basic** Integer array returned to a **VBScript** script produces a value of 8196 (**vbInteger** + **vbArray**).
- To test for an array, you can use the intrinsic constant **vbArray**. For example:

```
If VarType(myVar) And vbArray Then
    MsgBox "This is an array"
End If
```

- Alternatively, you can also use the **IsArray** function.

Tips

- When you use **VarType** with an object variable, you may get what appears to be an incorrect return value. The reason for this is that if the object has a default property, **VarType** returns the data type of the default property.
- There is no such value as **vbNothing**.
- For most purposes, the **TypeName** function, which returns a string indicating a variable's data type, is much more convenient and easy to use.

TypeName Function

Description

The **TypeName** function returns a string containing the name of the data type of a variable.

Syntax

```
TypeName (varname)
```

Arguments

Parameter	Description
<i>varname</i>	Required. The name of a variable.

Return Value

A String.

Notes

- **TypeName** returns the variant's data type. If the variant has not been assigned a value, **TypeName** returns **Empty**. Therefore, **TypeName** never actually returns the string "Variant." systems.

IsDate Function

Description

The **IsDate** function returns a Boolean value indicating whether an expression can be converted to a date.

Syntax

```
IsDate (expression)
```

Arguments

Parameter	Description
<i>expression</i>	Can be any date expression or string expression recognizable as a date or time.

Return Value

Boolean (**True** or **False**).

Notes

- If the expression passed to **IsDate** is a valid date, **True** is returned; otherwise, **IsDate** returns **False**.
- In Microsoft Windows, the range of valid dates is January 1, 100 A.D. through December 31, 9999 A.D.; the ranges vary among operating systems.

Tips

- **IsDate** uses the locale settings of the current Windows system to determine whether the value held within the variable is recognizable as a date. Therefore, what is a legal date format on one machine may fail on another.
- **IsDate** is particularly useful for validating data input.

Example

The example, implements a **Select Case** branching, according to an output parameter.

```
Dim MyDate, YourDate, NoDate, InvalidDate1, InvalidDate2
Dim bRes
MyDate = "October 19, 1962"
YourDate = #29/10/62#
NoDate = "Hello"
InvalidDate1 = "40/40/2006"
InvalidDate2 = "February 31, 2000"
bRes = IsDate(MyDate)           ' returns True
bRes = IsDate(YourDate)        ' returns True
bRes = IsDate(NoDate)          ' returns False
bRes = IsDate(InvalidDate1)    ' returns False
bRes = IsDate(InvalidDate2)    ' returns False
```

Remarks

- On Long Dates formats, or date formats, the function depends on Regional

and Language settings. For example:

- Using the date "October 19, 1962" when the English(USA) format was set, the function **IsDate**("October 19, 1962") will return true, but, using other format, i.e. Hebrew, **IsDate**("October 19, 1962") will return **False**.

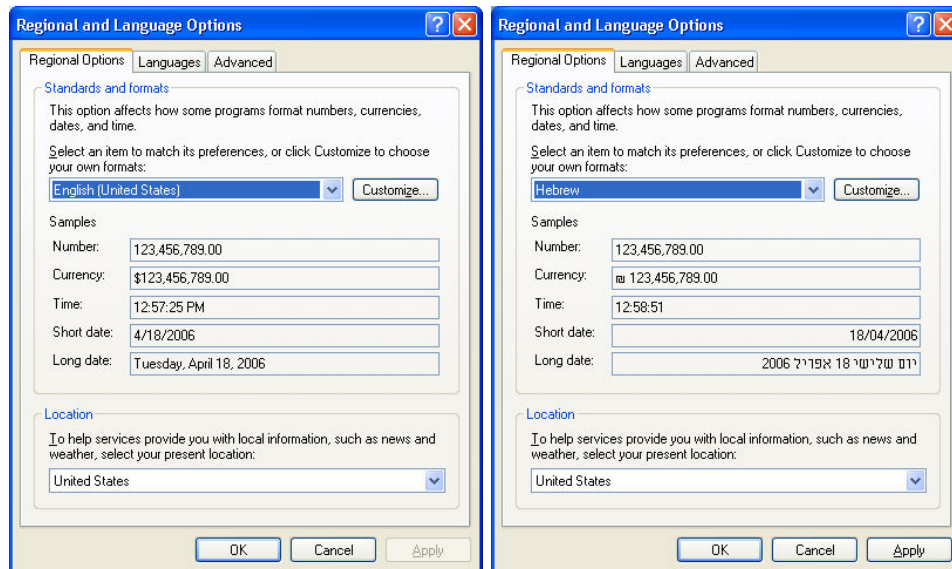


Figure 2 - English (USA) and Hebrew Formats

IsNumeric Function

Description

The **IsNumeric** function returns a Boolean value indicating whether an expression can be evaluated as a number.

Syntax

```
IsNumeric(expression)
```

Arguments

Parameter	Description
<i>expression</i>	can be any expression.

Return Value

Boolean (**True** or **False**).

Notes

- If the expression passed to **IsNumeric** evaluates to a number, **True** is returned; otherwise, **IsNumeric** returns False. systems.

Tips

- If expression is a date or time, **IsNumeric** evaluates to **False**.
- If expression is a currency value, including a string that includes the currency symbol defined by the Control Panel's Regional Settings applet,

IsNumeric evaluates to True.

Example

The following example uses the **IsNumeric** function to determine whether a variable can be evaluated as a number:

```
Dim bRes
nInteger =
bRes = IsNumeric(32)           ' returns True
bRes = IsNumeric("3$")        ' returns True
bRes = IsNumeric("1.44")      ' returns True
bRes = IsNumeric("1.4E-4")    ' returns False
bRes = IsNumeric("32A")       ' returns False
bRes = IsNumeric("1,234,567")  ' returns False/True
bRes = IsNumeric("1.234.567") ' returns False/True
bRes = IsNumeric(-456)        ' returns True
```

Remarks

- In English, the digit grouping symbol is a comma. Decimal Symbol is a dot, but in Spanish digit grouping symbol is a dot, and the decimal symbol is a comma or a dot.
- **IsNumeric** can return **True** or **False**, depending on the local machine regional settings and Regional Options Settings.

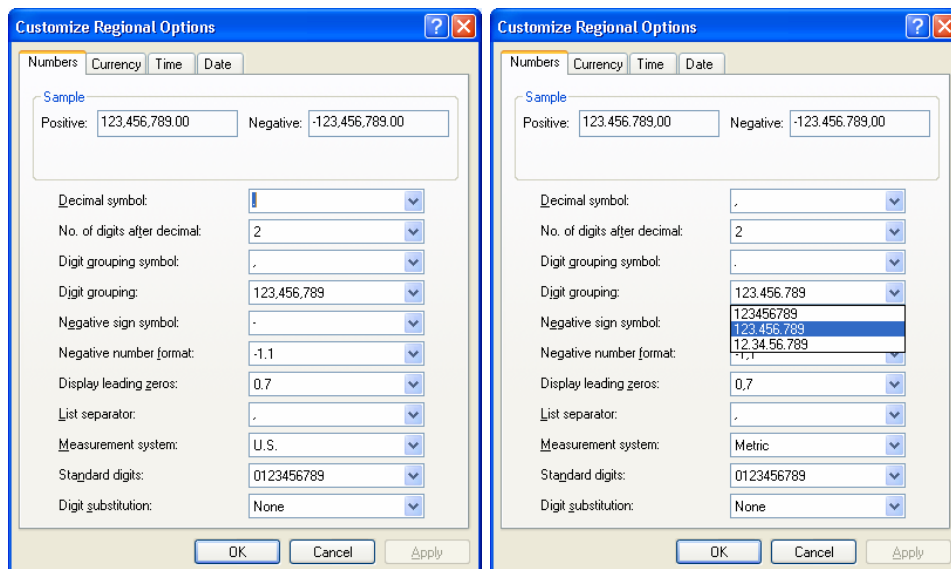


Figure 3 – Regional Options English (USA) and Spanish (Argentina)

IsEmpty Function

Description

The **IsEmpty** function returns a **Boolean** value indicating whether an expression can be evaluated as a number.

Syntax

```
IsEmpty(expression)
```

Arguments

Parameter	Description
<i>expression</i>	can be any expression.

Return Value

Boolean (**True** or **False**).

Notes

- If the variant passed to **IsEmpty** has not been initialized, **True** is returned; otherwise, **IsEmpty** returns **False**.
- Although **IsEmpty** can take an expression as the value of varname, it always returns **False** if more than one variable is used in the expression. **IsEmpty** is therefore most commonly used with single variables.

Tips

- When passed an object variable that has been set equal to **Nothing**, the **IsEmpty** function returns **False**. Hence, the function should not be used to test whether a previously initialized object variable now holds a valid object reference.

Example

The following example uses the **IsEmpty** function to determine whether a variable has been initialized:

```
Dim vVar1, vVar2, vVar3, vVar4
Dim bRes
vVar1 = Null : vVar2 = 1
vVar3 = Empty
bRes = IsEmpty(vVar1)      ' returns False
bRes = IsEmpty(vVar2)      ' returns False
bRes = IsEmpty(vVar3)      ' returns True
bRes = IsEmpty(vVar4)      ' returns True
```

IsNull Function

Description

The **IsNull** function returns a **Boolean** value that indicates whether an expression contains no valid data (**Null**).

Syntax

```
IsNull(expression)
```

Arguments

Parameter	Description
<i>expression</i>	Can be any expression.

Return Value

Boolean (**True** or **False**).

Notes

- If the expression passed to **IsNull** is **Null**, **True** is returned; otherwise, **IsNull** returns **False**.
- All variables in expression are checked for null values. If a null value is found in any one part of the expression, True is returned for the entire expression.
- In **VBScript**, **Null** is a separate data type that can take on a single value, **Null**. It is used to indicate that data is missing. Because it represents missing data, all expressions that include a **Null** value also result in a **Null** value. This makes perfect sense. For instance, if we have an array containing two valid months of sales data and a **Null** representing the third month's sales data, the quarter's sales data should also be **Null**, since accurate data for the quarter is not available.

Tips

- **IsNull** is useful when returning data from a database. You should check field values in columns that allow Nulls against **IsNull** before assigning the value to a collection or other variable. This stops the common "Invalid Use of Null" error from occurring.
- **IsNull** is the only way to evaluate an expression containing a **Null**. For example, the seemingly correct statement:
If vMyVar = Null Then
- Always evaluates to **False**, even if *vMyVar* is **Null**. This occurs because the value of an expression containing **Null** is always **Null**, and therefore **False**.

Example

The following example uses the **IsNull** function to determine whether a variable contains a **Null**

```
Dim vNull, vVbNull, vVbNullString, vEmpty
Dim bRes
vNull = Null : vVbNull = vbNull : vVbNullStr = vbNullString
vEmpty = Empty
bRes = IsNull(vNull)           ' returns True
bRes = IsNull(vVbNull)        ' returns False
bRes = IsNull(vVbNullStr)     ' returns False
bRes = IsNull(vEmpty)         ' returns False
```

IsArray Function

Description

The **Array** function returns a variant array containing the elements whose values are passed to the function as arguments.

Syntax

```
IsArray(varname)
```

Arguments

Parameter	Description
<i>varname</i>	The name of the variable to be checked.

Return Value

Boolean (**True** or **False**).

Notes

- If the variable passed to **IsArray** is an array or contains an array, **True** is returned; otherwise, **IsArray** returns **False**.

Tips

- Due to the nature of variants, it isn't always obvious if a variant variable contains an array, especially if you pass the variant to a function, and the function may or may not attach an array to the variant.
- Calling any of the array functions — such as **LBound** or **UBound** - or trying to access an element in an array that doesn't exist will obviously generate an error. In these situations, you should first use the **IsArray** function to determine whether you can safely process the array.

Example

The following code demonstrates the usage of the array functions.

```
Option Explicit
Dim arrArray
Dim i
Dim sTmp
'--- Building the array.
arrArray = Array(81,117,105,99,107,84,101,115,116,45,80,114,111)
'--- Checking if variable is an array.
If IsArray(arrArray) Then
    For i = LBound(arrArray) To UBound(arrArray)
        arrArray(i) = Chr(arrArray(i))
    Next
End If
If IsArray(arrArray) Then
    sTmp = Join(arrArray, vbNullString)
    MsgBox sTmp
End If
Erase arrArray
```

IsObject Function

Description

The **IsObject** function returns a **Boolean** value indicating whether an expression references a valid Automation object.

Syntax

```
IsObject(expression)
```

Arguments

Parameter	Description
<i>expression</i>	Can be any expression.

Return Value

Boolean (**True** or **False**).

Notes

- If the variable passed to **IsObject** references or has referenced an object, even if its value is **Nothing**, **True** is returned; otherwise, **IsObject** returns **False**.

Tips

- **IsObject** doesn't validate the reference being held by an object variable; it simply determines whether the variable is an object. To ensure that an object reference is valid, you can use the syntax **Is Nothing**, as shown

```
If objVar Is Nothing Then
...
End if
```

Example

The following example uses the **IsObject** function to determine if an identifier represents an object variable:

```
Option Explicit
Dim oCreate, oGet, oSetObject, oSetRepository, oNothing
Dim vNoObject
Dim bRes
Set oCreate = CreateObject("Scripting.FileSystemObject")
'--- opens an excel application
SystemUtil.Run "c:\TableView.xls", "", "", ""
Set oGet = GetObject("c:\TableView.xls", "Excel.Workspace")
Set oSetObject = Browser("MSDN").Page("MSDNPage").Object
Set oSetRepository = Browser("MSDN").Page("MSDNPage")
Set oNothing = Nothing
vNoObject = Array(1,2,3)
bRes = IsObject(oCreate)           ' returns True
bRes = IsObject(oGet)             ' returns True
bRes = IsObject(oSetObject)       ' returns True
bRes = IsObject(oSetRepository)   ' returns True
bRes = IsObject(oNothing)         ' returns True
bRes = IsObject(vNoObject)        ' returns False
```

Is Nothing Expression

Description

- In **Visual Basic** and **VBA** editions the functions **IsNothing** and **IsMissing** are supported, but no in the **VBScript** edition.
- When scripting, we want to verify, that an object was set ok, or a function returned a valid object. The **Is** operator Compares two object reference variables.

- `result = object1 Is object2`; If `object1` and `object2` both refer to the same object, result is **True**; if they do not, result is **False**.
- So, comparing **Object** with **Nothing** is the supplement for the **IsNothing** function.
- You can add the follow function to your **vbs** libraries:

Example

```
Option Explicit
Public Function IsNothing( ByVal obj)
    If obj Is Nothing Then
        IsNothing = True
    Else
        IsNothing = False
    End If
End Function
```

Data Conversion Functions

CBool Function

Description

The **CBool** function returns an expression that has been converted to a **Variant** of subtype **Boolean**.

Syntax

```
CBool(expression)
```

Arguments

Parameter	Description
<i>expression</i>	Any numeric expression or a string representation of a numeric value.

Return Value

expression converted to a type of **Boolean** (**True** or **False**).

Notes

- Casts expression as a **Boolean** type. Expressions that evaluate to 0 are converted to **False** (0), and expressions that evaluate to nonzero values are converted to **True** (-1).
- If the expression to be converted is a string, the string must act as a number. Therefore, **CBool**("ONE") results in a type mismatch error, yet **CBool**("1") converts to **True**.

Tips

- You can check the validity of the expression prior to using the **CBool** function by using the **IsNumeric** function.
- When you convert an expression to a **Boolean**, an expression that evaluates to 0 is converted to **False** (0), and any nonzero number is converted to **True** (-1). Therefore, a **Boolean False** can be converted back to its original value (i.e., 0), but the original value of the True expression can't be restored unless it was originally -1.

Example

The following example uses the **CBool** function to convert an expression to a **Boolean**.

```
Dim a, b, c
Dim sTrue, sFalse
Dim sExpr1, sExpr2
Dim bRetVal
a = 5: b = 5 : c = -1
sTrue = "TRUE" : sFalse = "False"
sExpr1 = "qtp" : sExpr2 = "QTP"
bRetVal = CBool(a = b)
bRetVal = CBool(c) ' Returns True
bRetVal = CBool(sTrue) ' Returns True
bRetVal = CBool(sFalse) ' Returns False
bRetVal = CBool(StrComp(sExpr1, sExpr2, 0)) ' Returns True
bRetVal = CBool(StrComp(sExpr1, sExpr2, 1)) ' Returns False
```

CByte Function

Description

The **CByte** function returns an expression that has been converted to a **Variant** of subtype **Byte**.

Syntax

```
CByte(expression)
```

Arguments

Parameter	Description
<i>expression</i>	A string or numeric expression that evaluates between 0 and 255.

Return Value

expression converted to a type of Byte.

Notes

- Converts expression to a **Byte** data type. The **Byte** type is the smallest data storage device in **VBScript**. Being only one byte in length, it can store unsigned numbers between 0 and 255.
- If expression is a string, the string must be capable of being treated as a number.
- If expression evaluates to less than 0 or more than 255, an overflow error is

generated.

- If expression isn't a whole number, **CByte** rounds the number prior to conversion.

Tips

- Check that the value you pass to **CByte** is neither negative nor greater than 255.
- Use **IsNumeric** to insure the value passed to **CByte** can be converted to a numeric expression.
- When using **CByte** to convert floating-point numbers, fractional values up to but not including 0.5 are rounded down, while values greater than 0.5 are rounded up. Values of 0.5 are rounded to the nearest even number.

Example

The following example uses the **CByte** function to convert an expression to a byte:

```
Dim nSpanish, nEnglish
Dim nRetVal
nSpanish1 = "125,5678" : nEnglish1 = "125.5678"
nSpanish2 = "2,55" : nEnglish2 = "2.55"
'--- 1. Returns overflow error (Regional Settings "English USA")
'--- 2. Return 256 (Regional Settings "Spanish Argentina")
nRetVal = CByte(nSpanish1)
'--- 1. Returns overflow error (Regional Settings "Spanish Argentina")
'--- 2. Return 256 (Regional Settings "English USA")
nRetVal = CByte(nEnglish1)
'--- 1. Returns 255 (Regional Settings "English USA")
'--- 2. Returns 3 (Regional Settings "Spanish Argentina")
nRetVal = CByte(nSpanish2)
'--- 1. Returns 3 (Regional Settings "English USA")
'--- 2. Returns 255 (Regional
```

CDate Function

Description

The **CInt** function returns an expression that has been converted to a Variant of subtype Integer.

Syntax

CDate (*expression*)

Arguments

Parameter	Description
<i>expression</i>	Any valid date expression.

Return Value

expression converted into a **Date** type.

Notes

- **CDate** accepts both numerical date expressions and string literals. You can pass month names into **CDate** in either complete or abbreviated form; for example, "31 Dec 1997" is correctly recognized.
- You can use any of the date delimiters specified in your computer's regional settings; for most systems, this includes , / - and the space character.
- The oldest date that can be handled by the **Date** data type is 01/01/100, which in **VBScript** terms equates to the number -657434. Therefore, if you try to convert a number of magnitudes greater than -657434 with **CDate**, an error ("Type mismatch") is generated.
- The furthest date into the future that can be handled by the **Date** data type is 31/12/9999, which in **VBScript** terms equates to the number 2958465. Therefore, if you try to convert a number higher than 2958465 with **CDate**, an error ("Type mismatch") is generated.
- A "Type mismatch" error is generated if the values supplied in *expression* are invalid. **CDate** tries to treat a month value greater than 12 as a day value. If *expression* evaluates to less than 0 or more than 255, an overflow error is generated.
- If *expression* isn't a whole number, **CByte** rounds the number prior to conversion.

Tips

- Use the **IsDate** function to determine if *expression* can be converted to a date or time.
- A common error is to pass an uninitialized variable to **CDate**, in which case midnight will be returned.
- The **CDate** function can determine the day and month from a string regardless of their position, but only where the day number is larger than 12, which automatically distinguishes it from the number of the month. For example, if the string "30/12/97" were passed into the **CDate** function on a system expecting a date format of mm/dd/yy, **CDate** sees that 30 is obviously too large for a month number and treats it as the day. It's patently impossible for **CDate** to second guess what you mean by "12/5/97"—is it the 12th of May, or 5th of December? In this situation, **CDate** relies on the regional settings of the computer to distinguish between day and month. This can also lead to problems, as you may have increased a month value to more than 12 inadvertently in an earlier routine, thereby forcing **CDate** to treat it as the day value. If your real day value is 12 or less, no error is generated, and a valid, albeit incorrect, date is returned.
- If you pass a two-digit year into **CDate**, how does it know which century you are referring to? Is "10/20/97" 20 October 1997 or 20 October 2097? The answer is that two-year digits less than 30 are treated as being in the 21st Century (i.e., 29 = 2029), and two-year digits of 30 and over are treated as being in the 20th Century (i.e., 30 = 1930).
- Don't follow a day number with "st," "nd," "rd," or "th," since this generates a type mismatch error.
- If you don't specify a year, the **CDate** function uses the year from the current date on your computer.

Example

The following example uses the **CDate** function to convert a string to a date. In general, hard coding dates and times as strings (as shown in this example) is not recommended. Use date and time literals (such as #10/19/1962#, #4:45:23 PM#) instead.

```
Option Explicit
Dim sVeryLong, sLong, sShort, sTime
Dim dRetVal
sVeryLong = "Wednesday, April 19, 2006"
sLong = "April 19, 2006"
sShort = "4/19/2006" : sTime = "4:35:47 PM"
'--- Invalid format
If IsDate(sVeryLong) Then
    dRetVal = CDate(sVeryLong)
End If
dRetVal = CDate(sLong)           ' Returns #4/19/2006#
dRetVal = CDate(sShort)          ' Returns #4/19/2006#
dRetVal = CDate(sTime)           ' Returns #4:35:47 PM#
```

CInt Function

Description

The **CDate** function returns an expression that has been converted to a Variant of subtype Date.

Syntax

```
CInt(expression)
```

Arguments

Parameter	Description
<i>expression</i>	The range of expression is -32,768 to 32,767; fractions are rounded.

Return Value

expression cast as an integer type.

Notes

- *expression* must evaluate to a numeric value; otherwise, a type mismatch error is generated.
- If the value of expression is outside the range of the Integer data type, an overflow error is generated.
- When the fractional part of expression is exactly 0.5, **CInt** always rounds to the nearest even number. i.e. 0.5 rounds to 0, and 1.5 rounds to 2.
- Use the **CInt** function to provide internationally aware conversions from any other data type to an Integer subtype. For example, different decimal separators are properly recognized depending on the locale setting of your system, as are different thousand separators.

Tips

- Use **IsNumeric** to test whether expression evaluates to a number before

performing the conversion.

- **CInt** differs from the **Fix** and **Int** functions, which truncate, rather than round, the fractional part of a number. Also, **Fix** and **Int** always return a value of the same type as was passed in.

Example

The following example uses the **CInt** function to convert a value to an Integer

```
Option Explicit
Dim sNumber1, sNumber2, fSingle1, fSingle2, sHexa, sInvalid, sOverflow
Dim sScientific, sCurr
Dim nRetVal
sNumber1 = "4" : sNumber2 = "5" : fSingle1 = 3.14
fSingle2 = 4.5675675 : sHexa = &H2FFA : sInvalid = "2FFA"
sScientific = "3.1E3" : sCurr = "1$"
sOverflow = 1000000
If IsNumeric(sOverflow) Then
    nRetVal = CInt(sOverflow)           ' Run-time error
End If
'--- Invalid format
If IsNumeric(sInvalid) Then
    nRetVal = CInt(sInvalid)
End If
If IsNumeric(sHexa) Then
    nRetVal = CInt(sHexa)               ' Returns 12282
End If
nRetVal = CInt(sScientific)             ' Returns 3100
nRetVal = CInt(sCurr)                  ' Returns 1
nRetVal = CInt(sNumber1) + sNumber2    ' Returns 9
nRetVal = CInt(fSingle1)               ' Returns 3
nRetVal = CInt(fSingle2)
```

CLng Function

Description

The **CLng** function returns an expression that has been converted to a Variant of subtype Long.

Syntax

CLng (*expression*)

Arguments

Parameter	Description
<i>expression</i>	The range of expression is -2,147,483,648 to 2,147,483,647; fractions are rounded.

Return Value

expression cast as a type of **Long**.

Notes

- *expression* must evaluate to a numeric value; otherwise, a type mismatch error is generated.
- If the value of *expression* is outside the range of the long data type, an overflow error is generated.
- When the fractional part is exactly 0.5, **CLng** always rounds it to the nearest even number. For example, 0.5 rounds to 0, and 1.5 rounds to 2.
- Use **CInt** or **CLng** to force integer arithmetic in cases where currency, single-precision, or double-precision arithmetic normally would occur.
- Use the **CLng** function to provide internationally aware conversions from any other data type to a Long subtype. For example, different decimal separators are properly recognized depending on the locale setting of your system, as are different thousand separators.

Tips

- Use **IsNumeric** to test whether expression evaluates to a number.
- **CLng** differs from the **Fix** and **Int** functions, which truncate, rather than round, the fractional part of a number. Also, **Fix** and **Int** always return a value of the same type as was passed in.

Example

The following example uses the **CLng** function to convert a value to a Long

```
Option Explicit
Dim sNumber1, sNumber2, fSingle1, fSingle2, sHexa, sInvalid, sOverflow
Dim sScientific, sCurr
Dim nRetVal
sNumber1 = "4" : sNumber2 = "5" : fSingle1 = 3.14
fSingle2 = 4.5675675 : sHexa = &H2FFA : sInvalid = "2FFA"
sScientific = "3.1E3" : sCurr = "1$"
sOverflow = 10000000000
If IsNumeric(sOverflow) Then
    nRetVal = CLng(sOverflow)           ' Run-time error
End If
'--- Invalid format
If IsNumeric(sInvalid) Then
    nRetVal = CLng(sInvalid)
End If
If IsNumeric(sHexa) Then
    nRetVal = CLng(sHexa)               ' Returns 12282
End If
nRetVal = CLng(sScientific)            ' Returns 3100
nRetVal = CLng(sCurr)                  ' Returns 1
nRetVal = CLng(sNumber1) + sNumber2    ' Returns 9
nRetVal = CLng(fSingle1)               ' Returns 3
nRetVal = CLng(fSingle2)               ' Returns 5
```

CSng Function

Description

The **CSng** function returns an expression that has been converted to a Variant

of subtype **Single**.

Syntax

```
CSng (expression)
```

Arguments

Parameter	Description
<i>expression</i>	The range of expression is -3.402823E38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values.

Return Value

expression cast as a type of **Single**.

Notes

- *expression* must evaluate to a numeric value; otherwise, a type mismatch error is generated.
- If the value of *expression* is outside the range of the **Single** data type, an overflow error is generated.
- Use **CDbl** or **CSng** to force double-precision or single-precision arithmetic in cases where currency or integer arithmetic normally would occur.
- Use the **CSng** function to provide internationally aware conversions from any other data type to a **Single** subtype. For example, different decimal separators are properly recognized depending on the locale setting of your system, as are different thousand separators.

Tips

- Use **IsNumeric** to test whether *expression* evaluates to a number.
- If you need to use a floating-point number in **VBScript**, there is no reason to use a **Single**; use a **Double** instead. Generally, a **Single** is used because it offers better performance than a **Double**, but this is not true in **VBScript**. Not only is a **Single** not smaller than a **Double** in the **VBScript** implementation, but the processor also converts Singles to Doubles, performs any numeric operations, and then converts Doubles back to Singles.
- Test that *expression* evaluates to a number by using the **IsNumeric** function.

Example

The following example uses the **CSng** function to convert a value to a **Single**

```

Option Explicit
Dim sNumber1, sNumber2, fDouble1, fDouble2, sInvalid, sScientific, sMin, sCurr
Dim fRetVal
sNumber1 = "1.5" : sNumber2 = "4.11"
fDouble1 = 75.3421115: fDouble2 = 75.3421555
sScientific = "-1.4E-8"
sInvalid = "3000x" : sCurr = "300.5$"
sMin = "0.000000000345"
'--- Invalid format
If IsNumeric(sInvalid) Then
    fRetVal = CSng(sInvalid)
End If
If IsNumeric(sCurr) Then
    fRetVal = CSng(sCurr)           ' Returns 300.5
End If
fRetVal = CSng(sNumber1) + sNumber2 ' Returns 5.61
fRetVal = CSng(fDouble1)           ' Returns 75.34211
fRetVal = CSng(fDouble2)           ' Returns 75.34216
fRetVal = CSng(sScientific)         ' Returns -1.4E-08
fRetVal = CSng(sMin)

```

CDbl Function

Description

The **CSng** function returns an expression that has been converted to a **Variant** of subtype **Double**.

Syntax

```
CDbl(expression)
```

Arguments

Parameter	Description
<i>expression</i>	-1.79769313486232E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values.

Return Value

expression cast as a type of **Double**.

Notes

- *expression* must evaluate to a numeric value; otherwise, a type mismatch error is generated.
- If the value of *expression* is outside the range of the **Single** data type, an overflow error is generated.
- Use **CDbl** or **CSng** to force double precision or single-precision arithmetic in cases where currency or integer arithmetic normally would occur.
- Use the **CSng** function to provide internationally aware conversions from any other data type to a **Single** subtype. For example, different decimal separators are properly recognized depending on the locale setting of your

system, as are different thousand separators.

Tips

- Use **IsNumeric** to test whether expression evaluates to a number.
- If you need to use a floating-point number in **VBScript**, there is no reason to use a **Single**; use a **Double** instead. Generally, a **Single** is used because it offers better performance than a **Double**, but this is not true in **VBScript**. Not only is a **Single** not smaller than a **Double** in the **VBScript** implementation, but the processor also converts Singles to Doubles, performs any numeric operations, and then converts Doubles back to Singles.
- Test that expression evaluates to a number by using the **IsNumeric** function.

Example

The following example uses the **CSng** function to convert a value to a **Single**

```
Option Explicit
Dim fNum
Dim fRetVal
fNum = 234.456784
fRetVal = CDBl(fNum * 8.2 * 0.01) ' Returns 19.225456288
```

CCur Function

Description

The **CSng** function returns an expression that has been converted to a Variant of subtype Currency.

Syntax

```
CCur(expression)
```

Arguments

Parameter	Description
<i>expression</i>	A string or numeric expression that evaluates to a number between -922,337,203,685,477.5808 and 922,337,203,685,477.5807.

Return Value

expression converted to a type of **Currency**.

Notes

- If the expression passed to the function is outside the range of the **Currency** data type, an overflow error occurs.
- Expressions containing more than four decimal places are rounded to four decimal places.
- The only localized information included in the value returned by **CCur** is the decimal symbol.
- Use **CCur** to force currency arithmetic in cases where integer arithmetic normally would occur.
- You should use the **CCur** function to provide internationally aware conversions from any other data type to a **Currency** subtype. For example, different decimal separators and thousands separators are properly recognized depending on the locale setting of your system

Tips

- **CCur** doesn't prepend or append a currency symbol; for this, you need to use the **FormatCurrency** function. **CCur** does, however, correctly convert strings that include a localized currency symbol. For instance, if a user enters the string "\$1234.68" into a text box whose value is passed as a parameter to the **CCur** function, **CCur** correctly returns a currency value of 1234.68.
- **CCur** doesn't include the thousands separator; for this, you need to use the **FormatCurrency** function. **CCur** does, however, correctly convert currency strings that include localized thousands separators. For instance, if a user enters the string "1,234.68" into a text box whose value is passed as a parameter to the **CCur** function, **CCur** correctly converts it to a currency value of 1234.68.

Example

The example uses the **CCur** function to convert an expression to a **Currency**:

```
Option Explicit
Dim fUSD, fEUR, fYEN, fNIS, fLIR
Dim cRetVal
fUSD = "1000$" : fYEN = "1000¥" : fNIS = "1000₪"
fEUR = "100€" : fLIR = "1000£"
cRetVal = CCur(fUSD * 0.6) ' Returns 600 in Standard English USA format
cRetVal = CCur(fYEN * 0.6) ' Returns 600 in Standard Japan format
cRetVal = CCur(fNIS * 0.6) ' Returns 600 in Standard Hebrew Israel format
cRetVal = CCur(fEUR * 0.6) ' Returns 600 in French France format
cRetVal = CCur(fLIR * 0.6) ' Returns 600 in Standard English UK format
```

CStr Function

Description

The **CStr** function returns an expression that has been converted to a Variant of subtype String.

Syntax

```
CStr(expression)
```

Arguments

Parameter	Description
<i>expression</i>	Any expression that is to be converted to a string.

Return Values

If expression is	CStr returns
Boolean	A String containing True or False .
Date	A String containing a date in the short-date format of your system.
Null	A run-time error.
Empty	A zero-length String ("").
Error	A String containing the word Error followed by the error number.
Other numeric	String containing the number.

Notes

- Returns an expression that has been converted to a Variant of subtype String.
- Use **CStr** to force the result to be expressed as a String.
- You should use the **CStr** function instead of **Str** to provide internationally aware conversions from any other data type to a String subtype. For example, different decimal separators are properly recognized depending on the locale setting of your system.
- Almost any data can be passed to **CStr** to be converted to a string.
- Returns an expression that has been converted to a Variant of subtype String.
- Use **CStr** to force the result to be expressed as a String.
- You should use the **CStr** function instead of **Str** to provide internationally aware conversions from any other data type to a String subtype. For example, different decimal separators are properly recognized depending on the locale setting of your system.
- Almost any data can be passed to **CStr** to be converted to a string.

Array Functions

Erase Statement

Description

Reinitializes the elements of fixed-size arrays and deallocates dynamic-array storage space.

Syntax

```
Erase array
```

Arguments

Parameter	Description
<i>arraylist</i>	The name of the array/s variable to be erased.

Notes

- Specify more than one array to be erased by using commas to delimit *arraylist*.
- Fixed array variables remain dimensioned; on the other hand, all memory allocated to dynamic arrays is released.
- After the **Erase** statement executes, **TypeName** returns "Variant()" for a fixed-length array; in addition, the **IsEmpty** function returns **True** when individual members of the array are passed to it, and comparisons of an individual member of the array with an empty string ("") and with zero both return **True**.
- On the other hand, the **TypeName** function returns **Empty** for a dynamic array, and comparisons of the array with an empty string ("") and zero, also return **True**.

Tips

- Once you use **Erase** to clear dynamic arrays, they must be re-dimensioned with **ReDim** before being used again. This is because **Erase** releases the memory storage used by the dynamic array back to the operating system, which sets the array to have no elements.
- It is important to know whether an array is fixed-size (ordinary) or dynamic because **Erase** behaves differently depending on the type of array.

Example

The following example illustrates the use of the **Erase** statement

```
Dim NumArray(9)
Dim DynamicArray()
ReDim DynamicArray(9)      ' Allocate storage space.
Erase NumArray             ' Each element is reinitialized.
Erase DynamicArray         ' Free memory used by array.
```

UBound Function

Description

The **UBound** returns the largest available subscript for the indicated dimension of an array.

Syntax

```
UBound(array, dimension)
```

Arguments

Parameter	Description
<i>array</i>	Required. Name of the array variable; follows standard variable naming conventions.
<i>dimension</i>	Optional. Whole number indicating which dimension's upper bound is

	returned. Use 1 for the first dimension, 2 for the second, and so on. If dimension is omitted, 1 is assumed.
--	--

Return Value

Variant of type **Long**.

Notes

- If dimension is not specified, 1 is assumed. To determine the upper limit of the first dimension of an array created by **VBScript** code, set dimension to 1, set it to 2 for the second dimension, and so on.
- The upper bound of an array dimension can be set to any integer value using **Dim**, **Private**, **Public**, and **Redim**.

Tips

- Note that **UBound** returns the actual subscript of the upper bound of a particular array dimension.
- **UBound** is especially useful for determining the current upper boundary of a dynamic array.
- The **UBound** function works only with conventional arrays. To determine the upper bound of a collection object, retrieve the value of its Count or Length property.

LBound Function

Description

The **LBound** function returns the smallest available subscript for the indicated dimension of an array.

Syntax

```
LBound(array, dimension)
```

Arguments

Parameter	Description
<i>array</i>	Required. Name of the array variable; follows standard variable naming conventions.
<i>dimension</i>	Optional. Whole number indicating which dimension's upper bound is returned. Use 1 for the first dimension, 2 for the second, and so on. If <i>dimension</i> is omitted, 1 is assumed.

Return Value

Variant of type **Long** (0)

Notes

- If dimension isn't specified, 1 is assumed. To determine the lower limit of the first dimension of an array, set dimension to 1, to 2 for the second, and so on.

Tips

- This function appears to have little use in **VBScript**, since **VBScript** does not allow you to control the lower bound of an array. Its value, which is zero, is invariable. However, it is possible for **ActiveX** components created using **Visual Basic** to return an array with a lower bound other than zero to a **VBScript** script.
- **LBound** is useful when handling arrays passed by **ActiveX** controls written in **VB**, since these may have a lower bound other than zero.

Split Function

Description

The **Split** function returns a zero-based, one-dimensional array containing a specified number of substrings.

Syntax

```
Split(expression, delimiter, count, compare)
```

Arguments

Parameter	Description
<i>expression</i>	Required. String expression containing substrings and delimiters. If <i>expression</i> is a zero-length string, Split returns an empty array, that is, an array with no elements and no data.
<i>delimiter</i>	Optional. String character used to identify substring limits. If omitted, the space character (" ") is assumed to be the delimiter. If <i>delimiter</i> is a zero-length string, a single-element array containing the entire expression string is returned.
<i>count</i>	Optional. Number of substrings to be returned; -1 indicates that all substrings are returned.
<i>compare</i>	Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. The value can be one of the Comparison Constants values

Return Value

A variant array consisting of the arguments passed into the function.

Notes

- If *delimiter* isn't found in expression, **Split** returns the entire string in element 0 of the return array.
- If *delimiter* is omitted, a space character is used as the delimiter.
- If *count* is omitted or its value is -1, all strings are returned.
- The default comparison method is **vbBinaryCompare**. If *delimiter* is an alphabetic character, this setting controls whether the search for it in expression is case-sensitive (**vbBinaryCompare**) or not (**vbTextCompare**).
- Once count has been reached, the remainder of the string is placed, unprocessed, into the next element of the returned array.
- For a list of [Comparison Constants](#) see Table 6 on page 128

Join Function

Description

The **Join** function returns a string created by joining a number of substrings contained in an array.

Syntax

```
Join(list, delimiter)
```

Arguments

Parameter	Description
<i>list</i>	Required. One-dimensional array containing substrings to be joined.
<i>delimiter</i>	Optional. String character used to separate the substrings in the returned string. If omitted, the space character (" ") is used. If delimiter is a zero-length string, all items in the list are concatenated with no delimiters.

Return Value

A type **String**.

Notes

- If no delimiter is specified, the space character is used as a delimiter.
- The members of list (source array) must be convertible to strings. The individual members of list can be any data type except **Object**. In fact, the individual members of list can be objects as long as the object's default member is not another object. For example, the **Join** function in the code fragment:

```
Set fso = CreateObject("Scripting.FileSystemObject")
Set oDrive1 = fso.Drives("C")
Set oDrive2 = fso.Drives("D")

Set vArr(0) = oDrive1
Set vArr(1) = oDrive2

sJoin = Join(vArr, ",")
'--- returns the string "C:,D:".
```

- When a *delimiter* is specified, unused *list* elements are noted in the return string by the use of the *delimiter*. For example, if you specify a delimiter of "," and a source array with 11 elements, of which only the first two are used, Join returns a string similar to the following:
"a,b,,,,,,,,,"

Tips

The **Join** function is ideal for quickly and efficiently writing out a comma-delimited text file from an array of values.

Filter Function

Description

The **Filter** function Produces an array of matching values from an array of source values that either match or don't match a given filter string. In other words, individual elements are copied from a source array to a target array if they either match or do not match a filter string.

Syntax

```
Filter(InputStrings, Value, Include, Compare)
```

Arguments

Parameter	Description
<i>InputStrings</i>	Required. One-dimensional array containing substrings to be joined.
<i>Value</i>	Required. String to search for.
<i>Include</i>	Optional. Boolean value indicating whether to return substrings that include or exclude Value. If Include is True, Filter returns the subset of the array that contains Value as a substring. If Include is False, Filter returns the subset of the array that does not contain Value as a substring.
<i>compare</i>	Optional. Numeric value indicating the kind of string comparison to use.

Return Value

A **String** array of the elements filtered from *InputStrings*.

Notes

- The default *Include* value is **True**.
- The default Compare value is 0, **vbBinaryCompare**.
- For a list of [Comparison Constants](#) see Table 6 on page 128

Tips

- *InputStrings* elements that are **Empty** or that contain zero-length strings ("") are ignored by the **Filter** function.
- The array you declare to assign the return value of **Filter** should be a simple variant, as the following code fragment illustrates:

```
Dim aResult
aResult = Filter(sNames, sCriteria, True)
```

- Although the **Filter** function is primarily a string function, you can also filter numeric values. To do this, populate a *InputStrings* with numeric values. Although *Include* appears to be declared internally as a variant string, a **Long** or **Integer** can be passed to the function. For example:

```
Dim vSource, vResult
Dim sMatch

sMatch = CStr(2)
vSource = Array(10, 20, 30, 21, 22, 32)
vResult = Filter(vSource, sMatch, True, vbBinaryCompare)
```

- In this case, the resulting array contains four elements: 20, 21, 22, and 32.

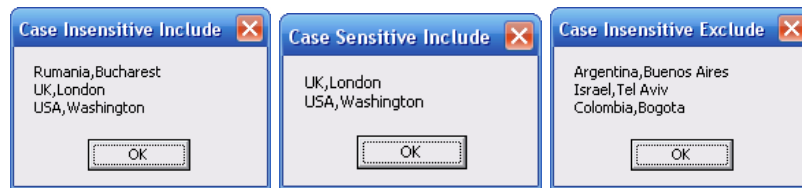
Example

The **Filter** function is an ideal companion to the **Dictionary** object. The **Dictionary** object is a collection-like array of values, each of which is stored with a unique string key. The **Keys** method of the **Dictionary** object allows you to produce an array of these Key values, which you can then pass into the **Filter** function as a rapid method of filtering the members of your **Dictionary**, as the following example demonstrates.

```
Option Explicit
Dim sKeys, sFiltered, sMatch, sMsg
Dim bInclude
Dim i
Dim oDict

Set oDict = CreateObject("Scripting.Dictionary")
oDict.Add "Argentina", "Buenos Aires"
oDict.Add "Israel", "Tel Aviv"
oDict.Add "Colombia", "Bogota"
oDict.Add "Rumania", "Bucharest"
oDict.Add "UK", "London"
oDict.Add "USA", "Washington"

sKeys = oDict.Keys
sMatch = "U" : bInclude = True
'--- find all keys that contain the string (case insensitive)
sFiltered = Filter(sKeys, sMatch, bInclude, vbTextCompare)
'--- now iterate through the resulting array
For i = 0 To UBound(sFiltered)
    sMsg = sMsg & sFiltered(i) & "," & oDict.Item(sFiltered(i)) & vbNewLine
Next
MsgBox sMsg, 0, "Case Insensitive Include"
sMsg = vbNullString
'--- find all keys that contain the character u (case sensitive)
sFiltered = Filter(sKeys, sMatch, bInclude, vbBinaryCompare)
'--- now iterate through the resulting array
For i = 0 To UBound(sFiltered)
    sMsg = sMsg & sFiltered(i) & "," & oDict.Item(sFiltered(i)) & vbNewLine
Next
MsgBox sMsg, 0, "Case Sensitive Include"
sMsg = vbNullString : bInclude = False
'--- find all keys that contain the character u (case sensitive)
sFiltered = Filter(sKeys, sMatch, bInclude, vbTextCompare)
'--- now iterate through the resulting array
For i = 0 To UBound(sFiltered)
    sMsg = sMsg & sFiltered(i) & "," & oDict.Item(sFiltered(i)) & vbNewLine
Next
MsgBox sMsg, 0, "Case Insensitive Exclude"
oDict.RemoveAll : Set oDict = Nothing
```



Array Function

Description

The **Array** function returns a variant array containing the elements whose values are passed to the function as arguments.

Syntax

```
Array(element1, element2, elementN,....)
```

Arguments

Parameter	Description
<i>element1</i>	The data to be assigned to the first array element.
<i>elementN</i>	Any number of data items you wish to add to the array.

Return Value

A variant array consisting of the arguments passed into the function.

Notes

- Although the array you create with the **Array** function is a variant array data type, the individual elements of the array can be a mixture of different data types.
- The initial size of the array you create is the number of arguments you place in the argument list and pass to the **Array** function.

The lower bound of the array created by the Array function is 0.

- The array returned by the **Array** function is a dynamic rather than a static array. Once created, you can re-dimension the array using **Redim**, **Redim Preserve**, or another call to the **Array** function.
- If you don't pass any arguments to the **Array** function, an empty array is created. Although this may appear to be the same as declaring an array in the conventional manner with the statement:

```
Dim myArray( )
```

- The difference is that you can then use the empty array with the **Array** function again later in your code.

Tips

- The **Array** function was not present in the first version of **VBScript** and was added to the language in Version 2.
- You cannot assign the return value of Array to a variable previously declared as an array variable. Therefore, don't declare the variant variable as an array using the normal syntax:

```
Dim MyArray( )
```

- Instead, simply declare a variant variable, such as:

```
Dim MyArray
```

- The **Array** function is ideal for saving space and time and for writing more efficient code when creating a fixed array of known elements, for example:

```
Dim arrTitles
arrTitle = Array("Mr", "Mrs", "Miss", "Ms")
```

- You can use the **Array** function to create multidimensional arrays. However, accessing the elements of the array needs a little more thought. The following code fragment creates a simple two-dimensional array with three elements in the first dimension and four elements in the second:

```
Dim vaListOne
vaListOne = Array( Array(1, 2, 3, 4), _
                  Array(5, 6, 7, 8), _
                  Array(9, 10, 11, 12))
```

- Surprisingly, the code you'd expect to use to access the array returns a "Subscript out of range" error:
- 'This line generates a Subscript out of range error

```
MsgBox vaListOne(1, 2)
```

- Instead, since this is an array stored within an array (that is, a ragged array), you can access it as follows:

```
MsgBox vaListOne(1)(2)
```

- Because you declare the variant variable to hold the array as a simple variant, rather than an array and can then make repeated calls to Array, the function can create dynamic arrays.
- For example, the following code fragment dimensions a variant to hold the array, calls Array to create a variant array, then calls Array again to replace the original variant array with a larger variant array:

```
Dim varArray
varArray = Array(10,20,30,40,50)
...
varArray = Array(10,20,30,40,50,60)
```

- The major disadvantage of using this method is that while it makes it easy to replace an array with a different array, it doesn't allow you to easily expand or contract an existing array.

Working with Objects

VBScript allows to create complex scripts using such advanced programming capabilities as branching, looping, error handling, and the calling of functions and subroutines. It does not, however, include intrinsic methods for performing system administration tasks. **VBScript** has built-in functions for determining the square root of a number or the **ASCII** value of a character, but no built-in methods for stopping services, retrieving events from event logs, or carrying out other tasks of

interest also to **QuickTest** programmers.

Fortunately, there are other ways to programmatically perform these tasks, primarily through the use of Automation objects. Automation objects are a subset of **COM (Component Object Model)**, a standard way for applications (.exe files) or programming libraries (.dll files) to present their functionality as a series of objects. In turn, programmers (or script writers) can use these objects, and the functionality of the application or programming library, in their own projects. For example, a word processing application might expose its spell checker as an Automation object, thus providing a way for script writers to add spell checking to their projects.

The ability to work with Automation objects and to utilize the properties and methods of these objects makes **VBScript** a powerful tool for system administration. Admittedly, **VBScript** alone cannot read events from an event log; however, **VBScript** can use the functionality included within **WMI** to retrieve such events. **VBScript** has no intrinsic methods for creating user accounts; however, the language can use the functionality in **ADSI** to create such accounts. In fact, **VBScript** is often referred to as a glue language because one of its primary uses is to "glue" objects together. Rather than provide a seemingly infinite number of intrinsic functions devoted to system administration, **VBScript** instead provides a framework for using the methods and properties of Automation objects designed to carry out these tasks.

Connecting to Objects

Before you can do anything with the data in a database, you must first make a connection of some kind to that database. Likewise, before you can do anything with the methods or properties of an Automation object, you must first make a connection to that object, a process known as binding.

Binding to objects can be confusing, because **VBScript** and **WSH** both provide a **GetObject** and a **CreateObject** method for accessing objects. Furthermore, although the implementations are similar, there are some subtle differences that grow in importance as you become more proficient in scripting. These differences are discussed in more detail later in this chapter. For now, use the following rules-of-thumb without worrying about whether you are using the **VBScript** or the **WSH** method (although in most cases you will use the **VBScript** implementation):

- Use **GetObject** to bind to either **WMI** or **ADSI**. Both **WMI** and **ADSI** allow you to use a moniker when binding from **VBScript**. A moniker (discussed later in this chapter) is an intermediary object that makes it easy to bind to objects that do not exist in the file system namespace. **ADSI** exists in the **Active Directory** namespace, while **WMI** exists in its own namespace.
- Use **CreateObject** to bind to all objects other than **WMI** or **ADSI**. In general, **CreateObject** will be required to create new instances of such elements as the **FileSystem** object, the **Dictionary** object, and **Internet Explorer**.

Creating an Object Reference

With Automation, you do not work directly with an object itself. Instead, you create a reference to the object by using **GetObject** or **CreateObject** and then assign this reference to a variable. After the reference has been created, you can

access the methods and properties of the object by using the variable rather than the object itself. Anytime you create an object reference, you must use the **Set** keyword when assigning the reference to a variable.

Set is a special **VBScript** statement that is used only when creating an object reference. If you use **Set** for other purposes, such as assigning a value to a variable, a run-time error will occur. For example, this line of code will fail because no object named 5 can be found on the computer:

```
Set x = 5
```

Calling Methods

Automation objects allow you to use their capabilities within your scripts. This enables you to create more powerful and more useful scripts than you could create if you were restricted to the functionality of the scripting language. For example, it is impossible to draw a chart or graph by using **VBScript** alone. Through Automation, however, you can borrow the capabilities of **Microsoft Excel** and easily add a chart or graph to, say, a Web page.

Automation objects typically expose both methods and properties. (However, there is no requirement for them to expose either.) Methods are equivalent to the actions that the object can perform. For example, Referenced objecta uses Automation to access the methods of two different **COM** objects and thus performs different actions

After you have created a reference to an object, you can call the methods of that object using dotnotation. Dot notation is so named because you call a method by typing the name of the variable that references the object, a period (or dot), and the name of the method. (Depending on the method, you might also type method parameters.) Generally, dot notation uses the following syntax:

```
ObjectReference.MethodName
```

Retrieving Properties

Properties are the attributes associated with an object. They are particularly important in system administration scripting because many of the objects you use are virtual representations of actual objects. For example

```
oLogicalDisk.FreeSpace
```

this object reference refers not to some amorphous programming construct but to an actual hard disk within the computer. The *FreeSpace* property is thus not just a property of an Automation object but also of drive C.

General Functions and Statements

Eval Function

Description

The **Eval** function evaluates an expression and returns the result.

Syntax

```
[result = ]Eval(expression)
```

Arguments

Parameter	Description
<i>result</i>	Optional. Variable to which return value assignment is made. If result is not specified, consider using the Execute statement instead.
<i>expression</i>	Required. String containing any legal VBScript expression.

Return Value

Any

Notes

- **Eval** follows the rules of precedence in evaluating expression. If an equals sign (=) occurs in expression, it is interpreted as a comparison operator rather than as an assignment operator.
- In this case, **Eval** returns True if the parts of expression are equal and **False** if they are not.

Example

In this example, the first result will always evaluate to **False**, since the variables are not equal, and the second will always evaluate to True, since *nTest1* is in fact less than *nTest2*:

```
Option Explicit
Dim nTest1, nTest2,
Dim vResult
nTest1 = 4 : nTest2 = 5
vResult = Eval("nTest1 = nTest2") ' Return False
vResult = Eval("nTest1 < nTest2") ' Return True
vResult = Eval("nTest1 / nTest2") ' Return 0.8
vResult = Eval("nTest1 - nTest2") ' Return 0.8
```

Tips

You may wonder why you'd want to bother with **Eval** when you can do the same thing without it. For example:

```
nVar1 = 2
nVar2 = 3
nResult = nVar1 + nVar2
```

is the same as:

```
nVar1 = 2
nVar2 = 3
nResult = Eval(nVar1 + nVar2)
```

But the significance of **Eval** is that it evaluates expressions stored to strings. For example, the code:

```
Dim sExp, result, a, b, c
a = 10
```

```
b = 20
c = 30
sExp = "a + b + c"
result = Eval(sExp)
```

Returns 60. This means that you can build expressions and assign them to strings dynamically, then have them evaluated by passing them to the **Eval** function.

RGB Function

Description

The **RGB** function returns a system color code that can be assigned to object color properties.

Syntax

```
RGB(red, green, blue)
```

Arguments

Parameter	Description
<i>red</i>	Required. A number between 0 and 255, inclusive.
<i>green</i>	Required. A number between 0 and 255, inclusive.
<i>blue</i>	Required. A number between 0 and 255, inclusive.

Return Value

A **Long** integer representing the **RGB** color value

Notes

- The **RGB** color value represents the relative intensity of the red, green, and blue components of a pixel that produces a specific color on the display.
- The **RGB** function assumes any argument greater than 255 is 255.
- The following table demonstrates how the individual color values combine to create certain colors:

Color	Red	Green	Blue
Black	0	0	0
Blue	0	0	255
Green	0	255	0
Red	255	0	0
White	255	255	255

Tips

- The **RGB** value is derived with the following formula:
RGB = red + (green * 256) + (blue * 65536)
- In other words, the individual color components are stored in the opposite order one would expect.
- **VBScript** stores the *red* color component in the low-order byte of the long

integer's low-order word, the *green* color in the high-order byte of the low-order word, and the *blue* color in the low-order byte of the high-order word.

- **VBScript** has a wide range of intrinsic color constants that can assign color values directly to color properties of objects.
- For a list of [Color Properties](#), see Table 9 on page 129

```
nVar1 = 2  
nVar2 = 3  
nResult = nVar1 + nVar2
```

GetLocale Function

Description

The **GetLocale** function returns the current locale ID value.

Syntax

```
GetLocale()
```

Return Value

A **Long** indicating the current **locale ID**.

Notes

- A locale **ID** represents a language as well as regional conventions. It determines such things as keyboard layout, alphabetic sort order, and date, time, number, and currency formats.

Tips

- If you want to temporarily change the locale, there is no need to call **GetLocale** and store its returned value before calling **SetLocale**, since **SetLocale** returns the value of the previous **locale ID**.
- **GetLocale** returns the **locale ID** currently in use by the script engine.
- Although you can set the locale using either a decimal, hexadecimal, or string **locale ID**, the **GetLocale** function returns only a decimal **locale ID** value.
- The default value of the script engine's **locale ID** is determined as follows: When the script engine starts up, the host passes it a **locale ID**. If the host does not do so, the script engine uses the user's default **locale ID**. If there is no user, then the script engine uses the system's default **locale ID**.
- Note that the script engine's **locale ID** is different from the system locale ID, the user **locale ID**, and the host application's **locale ID**. The **GetLocale** function reports the **locale ID** in use by the script engine only.
- For a list of valid [locale IDs](#) see – Locale ID's on page 136

SetLocale Function

Description

The **SetLocale** function sets the global locale and returns the previous locale.

Syntax

```
SetLocale (lcid)
```

Arguments

Parameter	Description
<i>lcid</i>	The <i>lcid</i> argument can be any valid 32-bit value or short string that uniquely identifies a geographical locale. Recognized values can be found in the registry under HKEY_CLASSES_ROOT\MIME\Database\Rfc1766. or HKEY_LOCAL_MACHINE\SOFTWARE\Classes\MIME\Rfc1766

Return Value

A Long indicating the previous locale ID.

Notes

- A **locale ID** represents language as well as regional conventions. It determines such things as keyboard layout, alphabetic sort order, and date, time, number, and currency formats.
- **Error! Reference source not found.**For a list of valid [locale IDs](#) see – Locale ID's on page 136
- If **SetLocale** is called with no arguments, it resets the script locale back to the host default, which is usually the user default.
- If *lcid* is zero or 1024, the locale is set as defined by the user's **locale ID**.
- If *lcid* is 2048, the local is set as defined by the system's regional settings.

Tips

- There is no need to call **GetLocale** and store its returned value before calling **SetLocale**, since **SetLocale** returns the value of the previous **locale ID**.
- **SetLocale** sets the **locale ID** of the script engine only. It does not affect the system, user, or host/application **locale IDs**.

Execute Statement

Description

The **Execute** statement executes one or more specified statements.

Syntax

```
Execute statement
```

Notes

- *statement* must evaluate to a string that contains one or more executable statements. An executable statement is any call to a user-defined procedure or function, or any intrinsic **VBScript** command.
- You can put multiple statements in the expression; separate them with colons.
- You can also separate the arguments with embedded line breaks.
- If statement includes an equal sign, it is interpreted as an assignment rather than an evaluation. For example, `x = 3` assigns the value 3 to the variable `x`, rather than comparing the value of the variable `x` with 3.
- In **VBScript**, a program fragment such as `x=3` can be interpreted as both an assignment statement (assigning the value 3 to the variable `x`) or as a comparison expression (for example `If x = 3 Then...`) The **Execute** and **ExecuteGlobal** statements always treat strings of the form `a = b` as assignment statements. Use **Eval** to interpret strings of this form as expressions.

Example

The following is a corrected version of an example appearing in online help that appears to do nothing. In this case, the **Execute** statement is used to execute a procedure named *Proc2*, and the entire source code for the procedure is also stored to the string *S* that is passed to the **Execute** statement:

```
Option Explicit
Dim S
S = "Proc2 : "
S = S & "Sub Proc2 : "
S = S & "Dim x : "
S = S & "x = 10 : "
S = S & "MsgBox X : "
S = S & "End Sub "
Execute S
```

But since the **Execute** statement only defines *Proc2* as a procedure that's visible within the script block but does not execute it, we must also execute *Proc2* as follows:

```
Option Explicit
Dim S
S = "Proc2 : "
S = S & "Sub Proc2 : "
S = S & "Dim x : "
S = S & "x = 10 : "
S = S & "MsgBox X : "
S = S & "End Sub "
Execute S
Proc2
```

Tips

- The **Execute** statement does for executable statements what the **Eval** function does for expressions: it allows you to dynamically (i.e., at runtime) assign code to a string and execute it by passing it to the **Execute** statement.
- Be careful with this technique, since it can lead to very hard-to-read code.

ExecuteGlobal Statement

Description

The **ExecuteGlobal** statement executes one or more specified statements in the global namespace of a script.

Syntax

```
ExecuteGlobal statement
```

Notes

- *statement* must evaluate to a string containing one or more executable statements. An executable statement is any call to a user-defined procedure or function, or to an intrinsic **VBScript** command.
- If *statement* contains multiple statements or lines of code, you can separate them with colons.
- You can also separate statements or lines of code with embedded line breaks (i.e., **vbCrLf**).
- If *statement* includes an equal sign, it is interpreted as an assignment rather than an evaluation. For example, `x = 3` assigns the value 3 to the variable `x`, rather than comparing the value of the variable `x` with 3.

Tips

- While the **Execute** statement executes code that inherits the scope of the procedure in which it was declared, **ExecuteGlobal** always executes code in the script's global scope. This has two major implications:
 - After the **ExecuteGlobal** statement runs, functions, procedures, or classes defined using **ExecuteGlobal** can be accessed from anywhere within the script.
 - Any variables accessed from code defined by the **ExecuteGlobal** statement must have global scope. In other words, when using **ExecuteGlobal** in a local scope, **ExecuteGlobal** will not see local variables.

LoadPicture Function

Description

The **LoadPicture** function returns a picture object. Available only on 32-bit platforms.

Syntax

```
LoadPicture (picturename)
```

Arguments

Parameter	Description
<i>picturename</i>	The path and filename of the picture file.

Return Value

A *StdPicture* object.

Notes

- *picturename* consists of an optional path along with the name of a supported image file. If the path component of *picturename* is omitted, the VBScript runtime engine attempts to find the image in the script's current directory.
- *picturename* can be a bitmap (.bmp), enhanced metafile (.emf), icon (.ico), Graphics Interchange Format (.gif), JPEG (.jpg), run-length encoded (.rle), or Windows metafile (.wmf).

Tips

The **StdPicture** object is defined by the **OLE** Automation library STDOLE2.TLB. It supports the members shown in the following table:

Name	Type	Description
<i>Handle</i>	Property	Returns a handle to the image.
<i>Height</i>	Property	Indicates the height of the image in HiMetric units.
<i>hPal</i>	Property	Returns a handle to the Picture object's palette.
<i>Render</i>	Method	Draws all or part of the image to a destination object.
<i>Type</i>	Property	Returns the Picture object's graphics format. Possible values are 0 (none), 1 (bitmap), 2 (metafile), 3 (icon), and 4 (enhanced metafile).
<i>Width</i>	Property	Indicates the width of the image in HiMetric units.

CreateObject Function

Description

The **CreateObject** function creates an instance of an **OLE** Automation (**ActiveX**) object. Prior to calling the methods, functions, or properties of an object, you are required to create an instance of that object. Once an object is created, you reference it in code using the object variable you defined.

Syntax

```
CreateObject (servername.typename, location)
```

Arguments

Parameter	Description
<i>servername</i>	The name of application providing the object.
<i>typename</i>	The programmatic identifier (ProgID) of the object to create, as defined in the system registry.
<i>location</i>	The name of the server where the object is to be created.

Return Value

A reference to an **ActiveX** object.

Notes

- Automation servers provide at least one type of object.
- For example, a word-processing application may provide an application object, a document object, and a toolbar object.
- To create an Automation object, assign the object returned by **CreateObject** to an object variable:

```
Dim oDoc
Set oDoc = CreateObject("Word.Document")
```

- Once an object is created, refer to it in code using the object variable you defined. you can access properties and methods of the new object using the object variable.
- Creating an object on a remote server can only be accomplished when Internet security is turned off.
- You can create an object on a remote networked computer by passing the name of the computer to the servername argument of **CreateObject**.
- That name is the same as the machine name portion of a share name.
- For a network share named "\\myserver\public", the servername is "myserver". In addition, you can specify servername using DNS format or an IP address.
- Programmatic identifiers use a string to identify a particular **COM** component or **COM**-enabled application. They are included among the subkeys of *HKEY_CLASSES_ROOT* in the system registry.
- If an instance of the **ActiveX** object is already running, **CreateObject** may start a new instance when it creates an object of the required type.
- Some common programmatic identifiers are shown in the following table:

ProgID	Description
Connection	An ActiveX Data Objects connection
ADODB.Recordset	An ActiveX Data Objects recordset
ADODB.Connection	An ActiveX Data Object connection
DAO.DBEngine	Data Access Objects
Internet.Explorer	An Internet Explorer Session.
Excel.Application	Microsoft Excel
Excel.Chart	A Microsoft Excel chart
Excel.Sheet	A Microsoft Excel workbook
MAPI.Session	Collaborative Data Objects
Outlook.Application	Microsoft Outlook
Scripting.Dictionary	Dictionary object
Scripting.FileSystemObject	File System object model
QuickTest.Application	QuickTest object model instance.

Tips

- In **Windows Script Host**, using the **CreateObject** method of the **WScript** object instead of the **VBScript CreateObject** function allows **WSH** to track the object instance and to handle the object's events.
- When using **VBScript** to develop an **Outlook** form, the **CreateObject** method of the Application object is the preferred way to instantiate an external class.
- **VBScript** offers the ability to reference an object on another network server.
- Using the *location* parameter, you can pass in the name of a remote server and the object can be referenced from that server. This means that you could even specify different servers depending upon prevailing circumstances, as this short example demonstrates:

```
Dim sMainServe
Dim sBackUpServer
sMainServer = "NTPROD1"
sBackUpServer = "NTPROD2"
If IsOnline(sMainServer) Then
    CreateObject("Sales.Customer",sMainServer)
Else
    CreateObject("Sales.Customer",sBackUpServer)
End If
```

- To use a current instance of an already running **ActiveX** object, use the **GetObject** function.
- If an object is registered as a single-instance object (i.e., an out-of-process **ActiveX** EXE), only one instance of the object can be created; regardless of the number of times **CreateObject** is executed, you will obtain a reference to the same instance of the object.
- Using the **CreateObject** function's location parameter to invoke an object remotely requires that the object be **DCOM**-aware. As an alternative, scripts can be run remotely using **Remote Windows Script Host**

GetObject Function

Description

The **GetObject** function returns a reference to an automation object. The **GetObject** function has three functions

- It retrieves references to objects from the Running Object Table.
- It loads persisted state into objects.
- It creates objects based on monikers.

Syntax

```
GetObject (pathname, class)
```

Arguments

Parameter	Description
<i>pathanme</i>	Optional. The full path and filename of a file that stores the state of an automation object, or a moniker (that is, a name that

	represents an object) along with the information required by the syntax of the moniker to identify a specific object.
<i>class</i>	Optional. The object's programmatic identifier (ProgID), as defined in the system registry.

Return Value

A reference to an **ActiveX** object.

Notes

- Although both pathname and class are optional, at least one argument must be supplied.
- **GetObject** can be used to retrieve a reference to an existing instance of an automation object from the Running Object Table. For this purpose, you supply the object's programmatic identifier as the class argument. However, if the object cannot be found in The Running Object Table, **GetObject** is unable to create it and instead returns runtime error 429, "ActiveX component can't create object." To create a new object instance, use the **CreateObject** function.
- If you specify a class argument and specify pathname as a zero-length string, **GetObject** returns a new instance of the object—unless the object is registered as single instance, in which case the current instance is returned. For example, the following code launches **Excel** and creates a new instance of the **Excel Application** object:

```
Dim oExcel
Set oExcel = GetObject("", "Excel.Application")
```

- In this case, the effect of the function is similar to that of **CreateObject**.
- To assign the reference returned by **GetObject** to your object variable, you must use the **Set** statement:

```
Dim myObject
Set myObject = GetObject("C:\OtherApp\Library.lib")
```

- To load an object's persisted state into an object, supply the filename in which the object is stored as the pathname argument and omit the *class* argument.
- The details of how you create different objects and classes are determined by how the server has been written; you need to read the documentation for the server to determine what you need to do to reference a particular part of the object. There are three ways you can access an **ActiveX** object:
 - The overall object library. This is the highest level, and it gives you access to all public sections of the library and all its public classes:

```
GetObject("C:\OtherApp\Library.lib")
```

- A section of the object library. To access a particular section of the library, use an exclamation mark (!) after the filename, followed by the name of the section:

```
GetObject("C:\OtherApp\Library.lib!Section")
```

- A class within the object library. To access a class within the library, use the optional *class* parameter:

```
GetObject("C:\OtherApp\Library.lib", "App.Class")
```

- To instantiate an object using a moniker, supply the moniker along with its required arguments. For details, see the discussion of monikers in the **Tips** section.

Tips

- Pay special attention to objects registered as single instance. As their type suggests, there can be only one instance of the object created at any one time. Calling **CreateObject** against a single-instance object more than once has no effect; you still return a reference to the same object.
- The same is true of using **GetObject** with a pathname of " "; rather than returning a reference to a new instance, you obtain a reference to the original instance of the object. In addition, you must use a pathname argument with single-instance objects (even if this is " "); otherwise an error is generated.
- You can't use **GetObject** to obtain a reference to a class created with **VBScript**; this can only be done using the **New** keyword.
- The following table shows when to use **GetObject** and **CreateObject** :

Use	Task
CreateObject	Create a new instance of an OLE server
CreateObject	Create a subsequent instance of an already instantiated server (if the server isn't registered as single instance)
GetObject	Obtain a further reference to an already instantiated server without launching a subsequent instance
GetObject	Launch an OLE server application and load an instance of a subobject
CreateObject	Instantiate a class registered on a remote machine
GetObject	Instantiate an object using a moniker

- A moniker is simply a name that represents an object without indicating how the object should be instantiated. (It contrasts with a programmatic identifier, for instance, which indicates that information stored in the system registry is used to locate and instantiate an object.) The following are some of the valid monikers recognized by the **GetObject** function, along with their required arguments:

Moniker	Arguments	Description
IIS:	metabasepath	Retrieves a reference to an IIS metabase object, which allows the programmer to view or modify the configuration of IIS
JAVA:	classname	Returns a reference to an unregistered Java object stored in the java\trustlib folder
SCRIPT:	path	Returns a reference to an unregistered Windows Script Component
CLSID:	clsid	Returns a reference to an object based on its class identifier (ClsID) in the system registry
WIMMGMTS:	string	Returns a reference to a WMI object that allows access to core Windows functionality

QUEUE:	clsid or progid	Uses MSMQ to return a reference to a queued COM+ component
NEW:	clsid or progid	Creates a new instance of any COM component that supports the IClassFactory interface (that is, of any createable COM component)

Example

```

Dim oIIS, oFS, oTxt
Dim sMsg, sFileName

fileName = "C:\IISClasses.txt"
' Creates an ISS moniker object
Set oIIS = GetObject ("IIS:// localhost")
IterateClasses oIIS, 0
' Creates a Scripting FileSystemobject object
Set oFS = CreateObject ("Scripting.FileSystemObject")
oTxt.Write sMsg
oTxt.Close
MsgBox "IIS Metabase information written to " & filename

Private Sub IterateClasses (oColl, nIndent)
    Dim oItem
    For Each oItem In oColl
        sMsg = sMsg & space(nIndent) & oItem.Name & vbCrLf
        IterateClasses oItem, nIndent + 3f
    Next
End Sub

```

GetRef Function

Description

The **GetRef** function returns a reference to a sub or function. This reference can be used for such purposes as binding to events or defining callback functions.

Syntax

```
GetRef (procname)
```

Arguments

Parameter	Description
<i>procname</i>	Required. Name of a sub or function

Return Value

A **Long** containing a reference to *procname*.

Notes

- **GetRef** can be used whenever a function or procedure reference is expected.
- When using **GetRef** to define event handlers for events, the **Set** keyword is required.

```
Set Web.OnLoad = GetRef("ShowGreetingDialog")
```

Tips

- A common use of **GetRef** is to bind to **DHTML** events in Internet Explorer. You can use **GetRef** to bind to any of the events in the **DHTML** object model.
- **GetRef** can be used to pass the address of a procedure to a routine that expects the address of a callback function as an argument.

Call Statement

The Call Statement is described in details on [03 – VBScripts Basics](#) under [Subroutines and Function Procedures](#)

Set Statement

Description

The **Set** statement Assigns an object reference to a variable or property.

Syntax

```
Set objectvar = (objectexpression | New classname Nothing)
```

Arguments

Parameter	Description
<i>objectvar</i>	Required. The name of the object variable or property.
<i>objectexpression</i>	Optional. An expression evaluating to an object.
<i>New</i>	Optional Keyword. Creates a new instance of an object defined using the Class...End Class construct, or with the syntax New RegExp instantiates the Regular Expression object.
<i>classname</i>	Required. The name of the class defined by the Class...End Class construct to be instantiated.
<i>Nothing</i>	Optional Keyword Assigns the special data type Nothing to <i>objectvar</i> , thereby releasing the reference to the object.

Notes

- *objectvar* doesn't hold a copy of the underlying object; it simply holds a reference to the object.
- If the **New** keyword is used to instantiate a **VBScript** class defined using the **Class...End Class** construct, a new instance of the class is immediately created and its **Class** Initialize event fires. This applies only to classes defined using the **Class...End Class** construct.
- You can also instantiate a **Regular Expression** object with the **New**

keyword by using a statement like the following

```
Set oRegExp = New oRegExp
```

- If *objectvar* holds a reference to an object when the **Set** statement is executed, the current reference is released and the new one referred to in *objectexpression* is assigned.
- *objectexpression* can be any of the following:
 - The name of an object. This creates a duplicate object reference in which two references point to the same object.
 - A variable that has been previously declared and instantiated using the **Set** statement and that refers to the same type of object
 - A call to a function, method, or property that returns the same type of object.
- By assigning **Nothing** to *objectvar*, the reference held by *objectvar* to the object is released.

Tips

- You can have more than one object variable referring to the same object. However, bear in mind that a change to the underlying object using one object variable is reflected in all the other object variables that reference that object.
- For example, consider the following code fragment, in which the *oColorCopy* object reference is set equal to the *oColor* object:

```
Dim oColor, oColorCopy
Set oColor = New CColor ' CColor class not shown
Set oColorCopy = oColor
oColor.CurrentColor = "Blue"
Msgbox oColorCopy.CurrentColor
```

- Since both *oColor* and *oColorCopy* reference a single object, the value of the *CurrentColor* property is Blue in both cases.
- It is commonly believed that you should release object references as soon as you are finished with them using code like the following:

```
Dim myClass
Set myClass = New SomeObject
' Do something here
Set myClass = Nothing
```

- Most of the time, though, releasing object references is unnecessary, since they are released anyway by the garbage collector when the object reference goes out of scope. There are only a couple of situations in which it is necessary to explicitly release object references:
 - When the object encapsulates a scarce resource, such as a database connection. In this case, it often makes sense to release the object reference as soon as you are done with it.
 - When two objects hold references to one another. In this situation, the objects are not destroyed when their references go out of scope. And their references going out of scope means that it is no longer possible to release the objects programmatically. **VBScript** objects (i.e., objects instantiated from classes declared with the **Class... End Class** construct will be destroyed when the scripting engine is torn down, which may be

before application shutdown. **COM** objects instantiated with the **CreateObject** or **GetObject** functions, though, may persist until the application terminates.

- When trying to discover whether an object reference has been successfully assigned, you should determine if the object variable has been assigned as **Nothing**. However, you can't use the equality comparison operator (=) for this purpose; you must use the **Is** operator, as the following code snippet shows:

```
If objectvar Is Nothing Then
    ... 'assignment failed
End If
```

- Any function that returns an object reference requires the use of the **Set** statement to assign the reference to a variable.

ScriptEngine Function

Description

The **ScriptEngine** function Indicates the scripting language currently in use.

Syntax

```
ScriptEngine ()
```

Return Value

A String

ScriptEngineBuildVersion Function

Description

The **ScriptEngineBuildVersion** function returns the build number of the **VBScript** script engine.

Syntax

```
ScriptEngineBuildVersion ()
```

Return Value

A Long.

ScriptEngineMajorVersion Function

Description

The **ScriptEngineMajorVersion** function Indicates the major version (1, 2, etc.) of the scripting language currently in use.

Syntax

```
ScriptEngineMajorVersion ()
```


Return Value

A **Long**.

Tips

- The following table lists the versions of **VBScript** through 5.0, as well as the year in which they were released and the products with which they were initially released:

Version	Year	Product
1.0	1996	Internet Explorer 3.0
2.0	1997	IIS 2.0
3.0	1998	Internet Explorer 4.0, IIS 4.0, WSH 1.0, Outlook 98
4.0	1998	Visual Studio 6.0
5.0	1999	Internet Explorer 5.0
5.5	2001	Internet Explorer 5.5
5.6	2002	Microsoft Visual Studio .NET

ScriptEngineMinorVersion Function

Description

The **ScriptEngineMinorVersion** function Indicates the minor version (the number to the right of the decimal point) of the scripting language engine currently in use.

Syntax

```
ScriptEngineMinorVersion ()
```

Return Value

A **Long**.

Tips

- If your script requires some functionality available in a baseline minor version, you ordinarily would want to make sure that the script is running on that version or a later version. Test for a minor version with a code fragment like:

```
lMajor = ScriptingEngineMajorVersion()  
lMinor = ScriptingEngineMinorVersion()  
If (lMajor = 5 And lMinor >= 1) Or (lMajor > 5) Then
```

Dictionary Object

Scripts often retrieve information from an outside source, such as a text file or a database. After this information has been retrieved, it needs to be stored in memory so that the script can act upon it. Information such as this can be stored in individual variables (one variable for each bit of information) or in an array.

Alternatively, information can also be stored in a **Dictionary** object.

The **Dictionary** object functions as an associative array; that is, it stores values in key-item pairs. This is different from an array, which uses a numeric index to store values. The **Dictionary object** is similar to a **Collection object**, except that it's loosely based on the **Perl** associative array. Like an array or a **Collection object**, the **Dictionary object** holds elements, called items or members, containing data. A **Dictionary object** can contain any data whatsoever, including objects and other **Dictionary objects**. Access the value of these dictionary items by using unique **keys** (or named values) that are stored along with the data, rather than by using an item's ordinal position as you do with an array. This makes the **Dictionary object** ideal when you need to access data that is associated with a unique named value.

You can access each item stored to a **Dictionary object** by using the **For Each ...Next** construct. However, rather than returning a variant containing the data value stored to the **Dictionary object** as you would expect, it returns a variant containing the key associated with the member. You then have to pass this key to the **Item** method to retrieve the member, as the following example shows:

```
Dim vKey
Dim sItem, sMsg
Dim oDict
'--- Creating a dictionary object
Set oDict = CreateObject("Scripting.Dictionary")
'--- Adding items to dictionary
oDict.Add "One", "Engine"
oDict.Add "Two", "Wheel"
oDict.Add "Three", "Tire"
oDict.Add "Four", "Spanner"
'--- Looping the Collection
For Each vKey In oDict
    sItem = oDict.Item(vKey)
    sMsg = sMsg & sItem & vbCrLf
Next
MsgBox sMsg
```

Creating a Dictionary

Because the **Dictionary** is a **COM** object, it must be instantiated in the same fashion as any other **COM** Object. The following code statement creates an instance of the **Dictionary** object:

```
Set oDictionary = CreateObject("Scripting.Dictionary")
```

Configuring Dictionary Properties

The **Dictionary** object has only one configurable property, **Compare Mode**, which plays an important role in determining which keys can be added and which ones cannot. (It is also important in verifying which keys are present in a **Dictionary** and which ones are not.) By default, a **Dictionary** is created in binary mode, which means each key in the **Dictionary** is based on its **ASCII** value. This is important because the **ASCII** value of an uppercase letter is different from the **ASCII** value of that same lowercase letter. In binary mode, both of these services can be added to the **Dictionary** as individual keys:

- alerter
- ALERTER

In other words, with binary mode, you can inadvertently add multiple entries for the same item. Binary mode dictionaries can also be difficult to search. For example, if you try to verify the existence of the key Alerter, you will be told that the key does not exist; that is because no key exists with that same pattern of uppercase and lowercase letters. As a result, you might end up adding yet another key for this same item.

When a **Dictionary** is configured in text mode, uppercase and lowercase letters are treated identically. This helps eliminate duplicate keys; you cannot add a key for ALERTER if a key for alerter already exists. It is also much easier to verify the existence of a key; searching for either alerter or ALERTer will find the key named Alerter.

To configure the **Dictionary** mode, create an instance of the **Dictionary** object and then set the **CompareMode** property to one of the following values:

- 0 - Sets the mode to binary. This is the default value.
- 1 - Sets the mode to text.

```
Const TEXT_MODE = 1  
Set oDictionary = CreateObject("Scripting.Dictionary")  
oDictionary.CompareMode = TEXT_MODE
```

You cannot change the **CompareMode** property of a **Dictionary** if that **Dictionary** contains any elements. This is because the binary mode allows you to differentiate between keys based solely on uppercase and lowercase letters.

In text mode, however, these three keys are considered identical. If you had these elements in a binary **Dictionary** and were able to reconfigure that **Dictionary** in text mode, the **Dictionary** would suddenly contain three duplicate keys, and it would fail. If you must reconfigure the **Dictionary** mode, you first need to remove all items from the **Dictionary**.

Adding Key-Item Pairs to a Dictionary

After you have created an instance of the **Dictionary** object, you can use the **Add** method to add key-item pairs to the dictionary. The **Add** method requires two parameters, which must be supplied in the following order and separated by a comma:

- Key name
- Item value

Key	Value
Printer 1	Printing
Printer 2	Offline
Printer 3	Printing

```

Set oDictionary = CreateObject("Scripting.Dictionary")
oDictionary.Add "Printer 1", "Printing"
oDictionary.Add "Printer 2", "Offline"
oDictionary.Add "Printer 3", "Printing"

```

Dictionary keys must be unique. For example, the following two script statements will generate a run-time error because, after the first line is interpreted, the key "Printer 1" will already be in the **Dictionary**:

```

oDictionary.Add "Printer 1", "Printing"
oDictionary.Add "Printer 1", "Offline"

```

Inadvertently Adding a Key to a Dictionary

One potential problem in using the **Dictionary** object is that any attempt to reference an element that is not contained in the **Dictionary** does not result in an error. Instead, the non-existent element is added to the **Dictionary**. Consider the following script sample, which creates a **Dictionary**, adds three key-item pairs to the **Dictionary**, and then attempts to echo the value of a non-existent item, Printer 4:

```

Set oDictionary = CreateObject("Scripting.Dictionary")
oDictionary.Add "Printer 1", "Printing"
oDictionary.Add "Printer 2", "Offline"
oDictionary.Add "Printer 3", "Printing"
MsgBox oDictionary.Item("Printer 4")

```

When the script tries to echo the value of the nonexistent item, no run-time error occurs. Instead, the new key, Printer 4, is added to the **Dictionary**, along with the item value **Null**. As a result, the message box shown in Figure 4 appears.

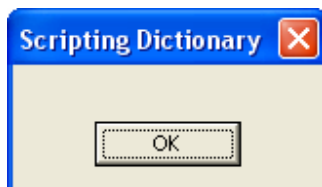


Figure 4 - Message Resulting from Inadvertently Adding a Key to a Dictionary

To avoid this problem, check for the existence of a key before trying to access the value of the item.

Manipulating Keys and Items in a Dictionary

By itself, a **Dictionary** is of little use; a **Dictionary** is valuable only when you can access, enumerate, and modify the keys and items within that **Dictionary**. After you have created a **Dictionary**, you will probably want to do such things as:

- Determine how many key-item pairs are in that Dictionary.
- Enumerate the keys and/or items within the Dictionary.
- Determine whether or not a specific key exists in the Dictionary.
- Modify the value of a key or an item in the Dictionary.
- Remove key-item pairs from the Dictionary.

All of these tasks can be carried out by using the methods and properties provided by the Script Runtime library.

Dictionary object, Properties and Methods

Dictionary.CompareMode Property

Description

The **CompareMode** property sets or returns the mode used to compare the keys in a **Dictionary object**.

Notes

- **CompareMode** can be set only on a dictionary that doesn't contain any data.
- The **CompareMode** property can have either of the following two values:

Value	Description
0, Binary	This is the default value. It compares the keys with a string byte-per-byte to determine whether a match exists.
1, Text	Uses a case-insensitive comparison when attempting to match keys with a string.

- In addition, the value of **CompareMode** can be greater than 2, in which case it defines the locale identifier (**LCID**) to be used in making the comparison.

Tips

- You need to explicitly set the **CompareMode** property only if you do not wish to use the default binary comparison mode.
- The Scripting Runtime type library defines constants (*BinaryCompare* and *TextCompare*) that can be used in place of their numeric equivalents. You can do this in one of three ways.
- You can use the equivalent **vbBinaryCompare** and **vbTextCompare** constants that are defined in the **VBScript** library.
- To see a list of [Comparison Constants](#) see Table 6 on page 128.
- Practically, the *CompareMode* property indicates whether the comparison between existing key names and the key argument of the **Dictionary** object's **Add** method, **Exists** method, **Item** property, or **Key** property will

be case-sensitive (*BinaryCompare*) or case-insensitive (*TextCompare*). By default, comparisons are case-sensitive.

Dictionary.Count Property

Description

The **Count** property is a read-only property that returns the number of **key/item** pairs in a **Dictionary object**.

Notes

- The **Count** property returns a **Long** value
- This property returns the actual number of items in the dictionary. So if you use the **Count** property to iterate the items in the dictionary, you would use code like the following:

```
Dim ctr
For ctr = 0 to dictionary.Count - 1
    ' do something
Next
```

Dictionary.Item Property

Description

The **Item** property sets or returns the data item to be linked to a specified key in a **Dictionary object**.

Syntax

```
dictionaryobject.Item(key) = item
```

Arguments

Parameter	Description
<i>key</i>	Required. A unique string key for this Dictionary object .
<i>item</i>	Optional. The data associated with <i>key</i> .

Notes

- The **Item** property is the default member of the **Dictionary object**.
- The data type is that of the item being returned.
- Unlike the **Item** property of most objects, the **Dictionary object's Item** property is read/write.
- If you try to set *item* to a nonexistent key, the key is added to the dictionary, and the item is linked to it as a sort of "implicit add."

Notes

- The **Dictionary object** doesn't allow you to retrieve an *item* by its ordinal position.
- If you provide a nonexistent key when trying to retrieve an item, the dictionary exhibits rather strange behavior: it adds key to the **Dictionary object** along with a blank item. You should therefore use the **Exists** method prior to setting or returning an item, as the example shows.

- If the item to be assigned or retrieved from the **Dictionary object** is itself an object, be sure to use the **Set** keyword when assigning it to a variable or to the **Dictionary object**.
- The comparison of key with member keys is defined by the value of the **Dictionary** object's **CompareMode** property.
- Although the read/write character of the **Dictionary** object's Item property has its drawbacks, it also has its advantages. In particular, it makes it easy to overwrite or replace an existing data item, since its Item property is read/write: simply assign the new value like you would with any other property.

Dictionary.Key Property

Description

The **Key** property replaces an existing key with a new one.

Syntax

```
dictionaryobject.Key(key) = newkey
```

Arguments

Parameter	Description
<i>key</i>	Required. The key of an existing dictionary <i>item</i> .
<i>newkey</i>	Required. A new unique key for this dictionary item.

Notes

- The **Key** property is write-only.
- *key*, the existing key value, must exist in the dictionary or an error results.
- *newkey* must be unique and must not already exist in the dictionary or an error results.
- The comparison of *key* and *newkey* with existing key values is defined by the **Dictionary** object's **CompareMode** property.

Tips

- Use the **Key** property to change the name of an existing key. Use the **Add** method to add a new key and its associated value to the **Dictionary object**. Use the **Keys** method to retrieve the names of all keys; this is especially useful when you don't know the names or the contents of the dictionary in advance.
- Attempting to retrieve the key name (a nonsensical operation, since this amounts to providing the key's name in order to retrieve the key's name) generates an error, as does attempting to modify a key name that hasn't already been added to the dictionary.
- Using a **For Each...Next** loop to iterate the members of a **Dictionary object** involves an implicit call to the Key property. In other words, each iteration of the loop returns a key, rather than a data item. To retrieve the member's data, you then must use its key value to access its data through the **Item** property. This is illustrated in the example for the **Dictionary.Item** property.

Dictionary.Add Method

Description

The **Add** method Adds a key and its associated item to the specified Dictionary object.

Syntax

```
dictionaryobject.Add (key, item)
```

Arguments

Parameter	Description
<i>key</i>	Required. A key value that's unique in the Dictionary object .
<i>item</i>	Required. The item to be added to the dictionary.

Notes

- If the key isn't unique, runtime error 457, "This key is already associated with an element of this collection," is generated.
- *item* can be of any data type, including objects and other **Dictionary** objects.

Tips

- The order of members within a **Dictionary object** is officially undefined. That is, you can't control the position of individual members, nor can you retrieve individual members based on their position within the **Dictionary object**. Your code, in short, should make no assumptions about the position of individual elements within the **Dictionary objects**.
- Once you add a key and its associated data item, you can change the key by using the write-only **Key** property.
- Use the **Dictionary object** to store tables of data, and particularly to store single items of data that can be meaningfully accessed by a key value.
- The use of **the Dictionary object** to store multifield data records is not recommended; instead, classes offer a better programmatic alternative. Typically, you would store a record by adding an array representing the record's field values to the dictionary. But assigning arrays to items in the **Dictionary object** is a poor programming practice, since individual elements of the array cannot be modified directly once they are assigned to the dictionary.

Dictionary.Exist Method

Description

The **Exist** method determines whether a given key is present in a **Dictionary object**.

Syntax

```
dictionaryobject.Exists (key)
```

Arguments

Parameter	Description
key	Required. The key value being sought.

Notes

- Returns **True** if the specified key exists in the **Dictionary** object; **False** if not.

Tips

- If you attempt to use the **Item** property to return the item of a nonexistent key, or if you assign a new key to a nonexistent key, the nonexistent key is added to the dictionary, along with a blank item. To prevent this, you should use the **Exists** property to ensure that the **Key** is present in the dictionary before proceeding.
- The way in which key is compared with the existing key values is determined by the setting of the **Dictionary** object's **CompareMode** property.

Dictionary.Items Method

Description

The **Items** method returns an array containing all the items in the specified **Dictionary object**.

Syntax

```
dictionaryobject.Items
```

Return Value

A **Variant** array.

Notes

- The returned array is always a zero-based variant array whose data type matches that of the items in the **Dictionary object**.

Tips

- The only way to directly access members of the **Dictionary** is via their key values.
- However, using the **Items** method, you can "dump" the data from the **Dictionary** into a zero-based variant array. The data items can then be accessed like an array in the normal way, as the following code shows:

```
Dim vArray
vArray = DictObj.Items
For i = 0 to DictObj.Count - 1
    MsgBox vArray(i) & vbNewLine
Next i
```

- The **Items** method retrieves only the items stored in a **Dictionary** object; you can retrieve all the **Dictionary** object's keys by calling its **Keys** method.

Dictionary.Keys Method

Description

The **Keys** method returns an array containing all the Key values in the specified **Dictionary object**.

Syntax

```
dictionaryobject.Keys
```

Return Value

An array of strings.

Notes

- The returned array is always a 0-based variant array whose data type is **String**.

Tips

- The **Keys** method retrieves only the keys stored in a **Dictionary object**. You can retrieve all the **Dictionary** object's items by calling its **Items** method. You can recall an individual data item by using the **Dictionary** object's **Item** property.

Dictionary.Remove Method

Description

The **Remove** method Removes both the specified key and its associated data (i.e., its item) from the dictionary.

Syntax

```
dictionaryobject.Remove key
```

Arguments

Parameter	Description
key	Required. The key associated with the item to be removed.

Notes

- If key doesn't exist, runtime error 32811, "Element not found," occurs.

Dictionary.RemoveAll Method

Description

The **RemoveAll** method clears out the dictionary; in other words, removes all keys and their associated data from the dictionary.

Syntax

```
dictionaryobject.RemoveAll
```

Return Value

An array of strings.

Notes

- If you want to remove a selected number of members rather than the entire contents of the dictionary, use the Remove method.

RegExp Object

More information about the **RegExp** object can be found in chapter

Class Object

Since **VBScript** 5.0, developers have been able to create classes to use in their scripts—a definite step along the road of object-oriented programming in **VBScript**. Writing classes with **VBScript** is very similar to writing **COM** objects with **VB**. Before we look at writing an actual class, let's go over some of the terminology so we are clear on what we are doing and what we are referring to.

A **class** is simply the template for an object. When you instantiate an object (that is, create an instance of a class) in code, **VBScript** makes a copy of the class for your use. All objects come from a class. Writing the class is simply a matter of creating a design for the objects that you want to use.

So naturally, it follows that an object is simply a copy of the class that you are making available to your program. You can make as many copies as you like for your use. The copies are temporary structures for holding information or creating interactions. When you are done with the objects, you can release them. If you need another one, you can instantiate another copy.

In **VBScript**, classes must be created in the scripts where you want to use them or they must be included in the scripts that use them. In **QuickTest**, classes can be created inside an action or in an external vbs file. Since **VBScript** isn't compiled, unless you use **Windows Script Components**, you don't have the advantage of being able to write a set of **VBScript COM** classes that are reusable outside of the scripts in which they're defined or that can be easily accessed by programs and scripts written in other languages.

The Class Construct

You declare a class using the **Class...End Class** construct. The syntax of the **Class** statement is:

```
Class classname
:
End Class
```

where classname is the name you want to assign to the class. It must follow standard **VBScript** variable naming conventions.

Classes can contain variables, properties, methods, and events. How many of these and of what types is completely up to you. It is possible to have an object that has no properties or methods and supports only the two default events, but it

won't be a very useful class.

The New Statement

To instantiate an object—that is, to create an instance of your class that you can use in your code—use the following syntax:

```
Set oClass = New classname
```

where oClass is the name you want to assign to your object variable (it again must follow standard **VBScript** variable naming conventions), and classname is the name of the class. The statement creates an object reference—that is, the variable oObj contains the address of your object in memory, rather than the object itself.

The Me Keyword

The **Me** Keyword represents the current instance of the class in which code is executing.

- **Me** is an implicit reference to the current object as defined by the **Class...End Class** statement.
- **Me** is automatically available to every procedure in a **VBScript** class.
- Values can't be assigned to the **Me** Keyword.
- The **Me** Keyword is particularly useful when passing an instance of the current class as a parameter to a routine outside of the class.

The Class Variables

In addition to properties, methods (which are either functions or subroutines), and events (which are subroutines), the code inside a **Class** structure can include variable definitions (but not variable assignments). The variable definition can take any of the following forms:

```
Dim varName1 [, varName2...]  
Private varName1 [, varName2...]  
Public varName1 [, varName2...]
```

The variable name must once again follow standard **VBScript** variable naming conventions.

The **Dim**, **Private**, and **Public** keywords indicate whether the variable is accessible outside of the class. By default, variables are public. They are visible outside of the **Class...End Class** structure. This means that the **Dim** and **Public** keywords both declare public variables, while the **Private** keyword declares a variable that's not visible outside of the class.

In general, it is **poor programming practice** to make a class variable visible outside of the class. There are numerous reasons for this, the most important of which is that you have no control over the value assigned to the variable (which is especially a problem when dealing with a weakly typed language like **VBScript**) and no ability to detect when the value of the variable has been changed. As a rule, then, all variables declared within your classes should be private.

The Class Properties

Typically, **class properties** are used to "wrap" the private variables of a class. That is, to change the value of a private variable, the user of your class changes the value of a property; the property assignment procedure (called a **Property Let** procedure) handles the process of data validation and assigning the new value to the private variable.

If the private variable is an object, use an object property assignment procedure (called a **Property Set** procedure) to assign the new property value to the private object variable. Similarly, to retrieve the value of a private variable, the user of your class retrieves the value of a property; the property retrieval procedure (called a **Property Get** procedure) handles the process of returning the value of the private variable.

Read-only properties (which wrap read-only private variables) have only a **Property Get** procedure, while write-only properties (which are rare) have only a **Property Let** or a **Property Set** procedure. Otherwise, properties have a **Property Get** procedure and either a **Property Let** or a **Property Set** procedure and are read-write.

Property Get Statement

Description

Declares the name, arguments, and code for a procedure that reads the value of a property and returns it to the calling procedure. The **Property Get** statement is used within a class defined by the **Class...End Class** construct.

Syntax

```
[Public [Default] | Private Property Get name [(arglist)]
    [statements]
    [name = expression]
[Exit Property]
    [statements]
    [name = expression]
End Property
```

Arguments

Arguments	Description
Public	Optional. Makes the property accessible from outside the class, giving it visibility through all procedures in all scripts. Public and Private are mutually exclusive.
Default	Optional. Used only with the Public keyword to indicate that a public property is the default property of the class.
Private	Optional. Restricts the visibility of the property to those procedures within the same Class...End Class code block. Public and Private are mutually exclusive.
name	Required. The name of the property.
arglist	Optional. A comma-delimited list of variables to be passed to the

	property as arguments from the calling procedure. arglist as the following syntax: [ByVal ByRef] argname[()]
ByVal	Optional. The argument is passed by value; that is, a local copy of the variable is assigned the value of the argument.
ByRef	Optional. The argument is passed by reference; that is, the local variable is simply a reference to the argument being passed. Changes made to the local variable are reflected in the argument. ByRef is the default way of passing variables.
argname	Required. The name of the local variable representing the argument.
statements	Optional. Program code to be executed within the property.
expression	Optional. The value to return from the property to the calling procedure.

Notes

- **Property** procedures are **Public** by default.
- The **Default** keyword indicates that this **Property Get** procedure is the class's default member. A default member is automatically executed in an assignment statement without the need to explicitly reference it. To take a common example, the following two statements are identical:

```
Set oCDrive = FileSystemObject.Drives.Item("C")
Set oCDrive = FileSystemObject.Drives("C")
```

- Both return a reference to a **Drive** object representing the local system's C drive. The second statement works because **Item** is the default member of the **Drives** collection.
- A class can have only a single default member. This must be a public procedure defined either by the **Property Get** statement or by the **Function** statement.
- Unlike other function and procedure names, the name of the **PropertyGetprocedure** doesn't have to be unique within its class module. Specifically, the **Property Let** and **Property Set** procedures can have the same name as the **Property Get** procedure. For example:

```
Property Let Name(sVal)
    msName = sVal
End Property
Property Get Name( )
    Name = msName
End Property
```

The number of arguments passed to a **Property Get** statement must match the corresponding **Property Let** or **Property Set** statement. For example:

```
Public Property Let MyProperty(sVal, nVal)
    miMyProperty = nVal
End Property
Public Property Get MyProperty(sVal)
    MyProperty = miMyProperty
End Property
```

- Both the **Property Let** and **Property Get** procedures share a common argument, sVal. The **Property Let** procedure has one additional argument, nVal, which represents the value that is to be assigned to the MyProperty property. (For details, see the next point.)
- In a **Property Let** procedure, the last argument defines the data assigned to the property. The data returned by the **Property Get** procedure must match the last argument of a corresponding **Property Let** or **Property Set** procedure.
- If an **Exit Property** statement is executed, the property procedure exits and program execution immediately continues with the statement from which the property procedure was called. Any number of **Exit Property** statements can appear in a **Property Get** procedure.
- If the value of the **Property Get** procedure has not been explicitly set when the program execution exits the procedure, its value will be empty, the uninitialized value of a variant.

Tips

- You can create a read-only property by defining a **Property Get** procedure without a corresponding **Property Let** or **Property Set** procedure.
- If the value of the property is an object, be sure to use the **Set** keyword when the **Property Get** procedure is called. For example:

```
Property Get Drive
    Set Drive = oDrive
End Property
```

- You should protect the value of properties by defining a Private variable to hold the internal property value and control the updating of the property by outside applications through the Property Let and Property Get statements, as the following template describes:

```
Class Object
    'Class Module Declarations Section
    'private data member only accessible from within this code module
    Private miMyProperty
    Public Property Let MyProperty(iVal)
        'procedure to allow the outside world to
        'change the value of private data member
        miMyProperty = iVal
        '(do not use a Property Let when creating a Read-Only Property)
    End Property
    Public Property Get MyProperty( )
        'procedure to allow the outside world to
        'read the value of private data member
        MyProperty = miMyProperty
    End Property
End Class
```

- Otherwise, if the variable used to store a property value is public, its value can be modified arbitrarily by any application that accesses the class containing the property.
- Using a **Property Let** procedure rather than allowing the user to access a class variable directly allows you to perform validation on incoming data.

For example, the following code insures that the value assigned to the Age property is a number that is between 0 and 110:

```
Class Person
  Private nAge

  Property Get Age
    Age = nAge
  End Property
  Property Let Age(value)
    ' Check that data is numeric
    If Not IsNumeric(value) Then
      Err.Raise 13      ' Type mismatch error
      Exit Property
    ' Check that number is in range
    ElseIf value < 0 Or value > 110 Then
      Err.Raise 1031    ' Invalid number error
    End If
    nAge = value
  End Property
End Class
```

- The default method of passing a parameter is **ByRef**, which means that any modifications made to a variable passed as an argument to the **Property Get** statement are reflected in the variable's value when control returns to the calling routine. If this behavior is undesirable, explicitly pass parameters by value using the **ByVal** keyword in *arglist*.
- You can use only the property defined by the **Property Get** statement on the right side of a property assignment.

Property Let Statement

Description

Declares the name, arguments, and code for a procedure that assigns a value to a property. The **Property Let** statement is used within a class defined by the **Class...End Class** construct.

Syntax

```
[Public | Private Property Let name ([arglist,] value)
  [statements]
  [Exit Property]
  [statements]
End Property
```

Arguments

Arguments	Description
Public	Optional Makes the property visible outside of the class, giving it visibility through all procedures in all scripts. Public and Private are mutually exclusive.
Private	Optional. Restricts the visibility of the property to those procedures within the same Class...End Class code block. Private and Public are

	mutually exclusive.
name	Required. The name of the property.
arglist	Optional. A comma-delimited list of variables to be passed to the property as arguments from the calling procedure. arglist as the following syntax: <code>[ByVal ByRef] varname[()]</code>
ByVal	Optional. The argument is passed by value; that is, a local copy of the variable is assigned the value of the argument.
ByRef	Optional. The argument is passed by reference; that is, the local variable is simply a reference to the argument being passed. Changes made to the local variable are reflected in the argument. ByRef is the default way of passing variables.
varname	Required. The name of the local variable containing either the reference or value of the argument.
value	The last (or only) argument in arglist; a variable containing the value to be assigned to the property.
statements	Optional. Program code to be executed within the property.

Notes

- A **Property Let** statement must contain at least one argument in *arglist*. If there is more than one argument, the last one contains the value to be assigned to the property. (This is the argument indicated as value in the prototype for the **Property Let** statement.)
- The last argument in *arglist* should correspond to both the private data member (at least, it should be defined as **Private**; used to hold the property value and the return value of the corresponding **Property Get** procedure, if there is one.
- **Property** procedures are **Public** by default.
- Unlike other functions and procedures, the name of the **Property Let** procedure can be repeated within the same module as the name of the **Property Get** and **Property Set** procedures.
- The number of the arguments passed to a **Property Let** statement must match the corresponding **Property Get** statement.
- If an **Exit Property** statement is executed, program flow continues with the statement following the call to the property. Any number of **Exit Property** statements can appear in a **Property Let** procedure.

Tips

- You should protect the values of properties by defining a **Private** variable to hold the internal property value and control the updating of the property by outside applications via **Property Let** and **Property Get**
- You can create a write-only property by defining a **Property Let** procedure without a corresponding **Property Get** procedure. Write-only properties, however, are comparatively rare, and are used primarily to prevent access to sensitive information such as passwords.
- The default method of passing parameters is **ByRef**, which means that any modifications made to a variable passed as an argument to the **Property Let** statement are reflected in the variable's value when control returns to the calling routine. If this behavior is undesirable, explicitly pass arguments

by value using the **ByVal** keyword in *arglist*.

- You can use the property defined by the **Property Let** statement only on the left side of a property assignment.

Property Set Statement

Description

Declares the name, arguments, and code for a procedure that assigns an object reference to a property. The **Property Set** statement is used within a class defined by the **Class...End Class** construct.

Syntax

```
[Public | Private Property Set name ([arglist,] reference)
    [statements]
    [Exit Property]
    [statements]
End Property
```

Arguments

Arguments	Description
Public	Optional. Makes the property visible outside of the class, giving it visibility through all procedures in all scripts. Public and Private are mutually exclusive.
Private	Optional. Restricts the visibility of the property to those procedures within the same Class...End Class code block. Private and Public are mutually exclusive.
name	Required. The name of the property.
arglist	Optional. A comma-delimited list of variables to be passed to the property as arguments from the calling procedure. arglist as the following syntax: [ByVal ByRef] <i>varname</i> [()]
ByVal	Optional. The argument is passed by value; that is, a local copy of the variable is assigned the value of the argument.
ByRef	Optional. The argument is passed by reference; that is, the local variable is simply a reference to the argument being passed. Changes made to the local variable are reflected in the argument. ByRef is the default way of passing variables.
varname	Required. The name of the local variable containing either the reference or value of the argument.
reference	Required. The last (or only) argument in arglist, it must be a variable containing the object reference to be assigned to the property.
statements	Optional. Program code to be executed within the property.

Notes

- A **Property Set** statement must contain at least one argument in arglist. (This is the argument indicated as reference in the statement's prototype.) If there is more than one argument, it's the last one that contains the object reference to be assigned to the property.

- The last argument in *arglist* must match both the private data member used to hold the property value and the data returned by the corresponding **Property Get** procedure, if there is one.
- **Property** procedures are **Public** by default.
- Unlike other variables and procedures, the name of a **Property Set** procedure can be repeated within the same module as the name of a **Property Get** procedure.
- The number of arguments passed to a **Property Set** statement must match the corresponding **Property Get** statement. For example:

```
Public Property Set MyProperty(nVal, oVal)
    Set miMyProperty(nVal) = oVal
End Property
Public Property Get MyProperty(nVal)
    Set MyProperty = miMyProperty(nVal)
End Property
```

- Both the **Property Set** and the **Property Get** procedures share a common argument, *nVal*. The **Property Set** procedure has one additional argument, *oVal*, which represents the object that is to be assigned to the *MyProperty* property.
- If an **Exit Property** statement is executed, program execution immediately continues with the statement following the call to the property. Any number of **Exit Property** statements can appear in a **Property Set** procedure.

Tips

- You should protect the values of properties by defining a **Private** variable to hold the internal property value and control the updating of the property by outside applications via **Property Set** and **Property Get** statements
- The default method of passing parameters is **ByRef**, which means that any modifications made to a variable passed as an argument to the **Property Set** statement are reflected in the variable's value when control returns to the calling routine. If this behavior is undesirable, explicitly pass arguments by value using the **ByVal** keyword in *arglist*.
- The property defined by the **Property Set** statement can occur only on the left side of a statement that assigns an object reference.

Using properties to wrap private variables

The use of public properties that are available outside of the class to wrap private variables is illustrated in the following example, which shows a simple class that defines a private variable, *modStrType*, and two read-write properties, *ComputerType* and *OperatingSystem*, the latter of which is an object property. Normally, you would validate the incoming data in the **Property Let** and **Property Set** procedures before assigning it to private variables, although that hasn't been done here to keep the example as simple as possible.

```
Class Computer
    Private modStrType
    Private oOS

    Public Property Let ComputerType(sType)
        modStrType = sType
```

```
End Property
Public Property Get ComputerType( )
    ComputerType = modStrType
End Property
Public Property Set OperatingSystem(oObj)
    Set oOS = oObj
End Property
Public Property Get OperatingSystem( )
    Set OperatingSystem = oOS
End Property

End Class
```

The Class Methods

Methods allow the class to do something. There is no magic to methods; they are simply subroutines or functions that do whatever it is you wish for the object to do. For example, if we created an object to represent a laptop computer in a company's inventory, then we would like to have a method that reports the laptop's owner. The Example shows a class with such a method.

Example: Creating a class method

```
Class LaptopComputer
    Private modOwner

    Public Property Let CompOwner(strOwner)
        modOwner = strOwner
    End Property
    Public Property Get CompOwner( )
        CompOwner = modOwner
    End Property
    Public Function GetOwner( )
        GetOwner = modOwner
    End Function
End Class
```

As with properties, you can use the **Public** and **Private** keywords to make methods available inside or outside of the **class**. In the previous example, the method and both properties are available outside of the **class** because they are declared as **Public**.

Note that in the Example, the **Property Get** procedure performs the same functionality as the **GetOwner** method. This is quite common: you often can choose whether you want to implement a feature as a property or as a method. In this case, you could define both property procedures to be private; then the only way for anyone to get the owner information from the object would be to invoke the **GetOwner** method.

The **GetOwner** method is declared as a function because it returns a value to the calling code. You can write methods as subroutines as well. You would do this when the method that you are calling does not need to pass back a return value to the caller.

The Class Events

Two events are automatically associated with every class you create:

Class_Initialize and **Class_Terminate**. **Class_Initialize** is fired whenever you instantiate an object based on this class. Executing the statement:

```
Set objectname = New classname
```

Causes the event to fire. You can use this event to set class variables, to create database connections, or to check to see if conditions necessary for the creation of the object exist. You can make this event handler either public or private, but usually event handlers are private; this keeps the interface from being fired from outside code. The general format of the **Class_Initialize** event is:

```
Private Sub Class_Initialize( )  
    Initialization code goes here  
End Sub
```

The **Class_Terminate** event handler is called when the script engine determines that there are no remaining references on an object. That might happen when an object variable goes out of scope or when an object variable is set equal to Nothing, but it also might not happen at either of these times if other variables continue to refer to the object. You can use this handler to clean up any other objects that might be opened or to shut down resources that are no longer necessary. Consider it a housekeeping event. This is a good place to make sure that you have returned all memory and cleaned up any objects no longer needed. The format of the **Class_Terminate** event is:

```
Private Sub Class_Terminate( )  
    Termination code goes here  
End Sub
```

Once again, the event handler can either be public or private, though ordinarily it's defined as private to prevent termination code from being executed from outside of the class.

Initialize Event

Description

Use the **Initialize** event of a class defined with the **Class...End Class** construct to prepare the object or class for use, setting any references to sub-objects or assigning default values to properties and values to class-level variables.

Syntax

```
Private Sub Class_Initialize( )  
End Sub
```

Notes

- The Initialize event is triggered automatically when a class is first instantiated by the **Set** statement. For example, in the following code, the Set statement generates the Initialize event:

```
Dim MyObject
'some code
...
'initialize event called here
Set MyObject = New MyClass
sName = MyObject.CustName
```

- The Initialize event doesn't take any arguments.
- It is best to declare the Initialize event as **Private**, although this is not required.

Tips

- While it's possible to explicitly call the Initialize event from within the object at any stage after the object has been created, it isn't recommended, because the code in the Initialize event should be written to be "run once" code.
- Use the Initialize event of a class module to generate references to dependent objects.

Terminate Event

Description

The **Terminate** event is fired when an instance of a class is removable from memory.

Syntax

```
Private Sub Class_Terminate( )
End Sub
```

Notes

- The **Terminate** event applies to classes defined with the **Class...End Class** construct.
- Instances of a class are removed from memory by explicitly setting the object variable to **Nothing** or by the object variable going out of scope.
- If a script ends because of a runtime error, a class's Terminate event isn't fired.

Tips

- Because the **Terminate** event is fired when an object becomes removable from memory, it is possible, but not recommended, for the **Terminate** event handler to add references back to itself and thereby prevent its removal. However, in this case, when the object actually is released, the Terminate event handler will not be called again.
- The Terminate event is fired under the following conditions:
 - An object goes out of scope.
 - The last reference to an object is set equal to **Nothing**.
 - An object variable is assigned a new object reference.
- The **Terminate** event is fired when an object is about to be removed from memory, not when an object reference is about to be removed. In other words, if two variables reference the same object, the **Terminate** event will be fired only once, when the second reference is about to be destroyed.

Example

- The following example shows a typical **Terminate** event in a class object that decrements a global instance counter used to ensure that only a single instance of a particular utility object is created. When the counter reaches 0, the global object reference to the utility object is destroyed.

```
Private Sub Class_Terminate( )  
    glbUtilCount = glbUtilCount - 1  
    If glbUtilCount = 0 then  
        Set goUtils = Nothing  
    End If  
End Sub
```

Classes in QuickTest

QuickTest supports Class objects. **Classes** usage in **QuickTest** not recommended. The reasons, maybe, are not what you expect. The **first** reason is, that is very hard to maintenance. While development environments provide tools to work with classes, **QuickTest** does not.

In Visual Studio, for example, you have a class tree view, you can search and navigate between methods, properties etc. in **QuickTest**, you will have a big large code. And large code, is difficult to maintenance. The **second** reason is the human Programming level. Not everyone knows how to work with classes. **QuickTest** designed to work with basic developers. Moreover, classes....

But, if you are an advanced user, and you know what is **Object Oriented Programming**, you will find no difficult to use classes (you can develop your class in VB-Visual Studio then copy your code to a **vbs** file, after making compatibility changes for **VBScript**).

Class Demo: DataTableUtil

Class Demo: Win32Directory

For more information see:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wmisdk/wmi/win32_directory.asp

FileSystemObject Object

More information about the **FileSystemObject** object can be found in chapter

Err Object

More information about the **Err** object can be found in chapter

COM Objects

The **Component Object Model (COM)** provides a standard way for applications (.exe files) or libraries (.dll files) to make their functionality available to any **COM**-compliant application or script. That is the textbook definition of **COM**. What **COM** really does, however, is make it possible for nonprogrammers to write scripts for managing Windows operating systems. **COM** provides a mechanism for translating script code into commands that can be acted on by the operating system. Without **COM**, anyone hoping to automate system administration would have to master not only a high-level programming language such as C++, VB.NET, C# or Visual Basic but also all of the Windows **Application Programming Interfaces (APIs)**. In effect, **COM** brings Windows programming to the masses.

COM components are files (typically .exe or .dll files) that contain definitions of the objects the component has available for use. (These definitions are known as classes.) When you create a **COM** object in a script (a process known as instantiation), you are creating an instance, or copy, of one of the classes contained within the **COM** component. After the instance has been created, you can then take advantage of the properties, methods, and events exposed by the object.

Objects that make their functionality available through **COM** are known as **COM** servers. Applications or scripts that make use of that functionality are referred to as **COM** clients. For example, when you write a script that uses **WMI**, **WMI** is the **COM** server and your script is the **COM** client. **COM** servers can be implemented in one of two ways:

- Out-of-process servers. Out-of-process servers are typically implemented in executable files and run in a different process than the script. For example, when you start a script, an instance of Wscript.exe begins to run. If you next instantiate a **Microsoft Word** object, you are then working with two processes: Wscript.exe and the Winword.exe process in which the **Microsoft Word** object runs.
- In-process servers. Libraries (.dll files) are known as in-process servers because they run in the same process as the application or script that called them. For example, when you call the **FileSystemObject** from within a script, no new process is created. This is because the **FileSystemObject** (which is found in the Sccrun.dll library) is an in-process server and thus runs in the same process as the script. In-process servers typically run faster than out-of-process servers because the operating system does not have to cross process boundaries (from the script process to the object process and then back) to access the objects methods and properties.

As noted previously in this chapter, **VBScript** works with a subset of objects known as Automation objects. All **COM** objects must support one or more interfaces, which are simply the avenues by which a **COM** client can access the **COM** server. Any object that supports the **IDispatch** interface is known as an Automation object. Because not all **COM** objects support the **IDispatch** interface, **VBScript** cannot access all of the **COM** objects on your computer.

The COM Process

As a script writer, you have to know only how to create a reference to an Automation object. You do not have to worry about how to locate and load the object because the Windows operating system takes care of that for you. Nevertheless, it is still useful for you to understand what happens between the time when the script runs your **CreateObject** command and the time when the object itself is available for use.

What is especially useful to understand is that there is no magic here; in fact, the process is relatively simple. When you install an application or a library that contains object classes, that application or library registers itself with the operating system, a procedure that enables the operating system to know:

- That a new **COM** server is available for use.
- The object classes that the new **COM** server makes available.

The registration process includes adding a number of subkeys to **HKEY_CLASSES_ROOT** in the registry. Among the subkeys that are added is one that specifies the Programmatic Identifier (**ProgID**) for each new object class. The **ProgID** is a short text string that identifies the name given to each object class. In addition, the **ProgID** is the parameter used in the **CreateObject** and **GetObject** calls to specify the object you want to create. For example, in the following line of code, the **ProgID** is **Excel.Application**:

```
Set oExcelApp = CreateObject("Excel.Application")
```

The **ProgID** is all the information the operating system needs to locate and instantiate the **COM** object.

Creating a New Object

When a **CreateObject** call runs, the script engine parses out the **ProgID** and passes that to a **COM API**. This **API** actually creates the object reference. For example, in this line of code, the string **Scripting.FileSystemObject** is passed to the **COM API**:

```
Set oFso = CreateObject("Scripting.FileSystemObject")
```

The **COM API** searches the **HKEY_CLASSES_ROOT** portion of the registry for a subkey with the same name as the **ProgID**. If such a subkey is found, the **API** then looks for a subkey named **CLSID**.

The **CLSID** subkey maintains a globally unique identifier (**GUID**) for the Automation object being created. The **GUID** will look something like this:

```
{172BDDF8-CEEA-11D1-8B05-00600806D9B6}
```

The **GUID** is the way that the operating system tracks and uses **COM** objects. The **ProgID** is simply an alias that is easier for script writers to remember.

After the **GUID** is discovered, the

HKEY_LOCAL_MACHINE\Software\Classes\CLSID portion of the registry is searched for a subkey with the same name as the **GUID**. When the operating system finds this subkey, it examines the contents for additional subkeys that store the information needed to locate the executable file or library file for the object (in the case of the **FileSystemObject**, C:\Windows\System32\Scrrun.dll). The **COM API** loads the application or library, creates the object, and then returns an object reference to the calling script.

Server Mode

When an object is created from an executable file, the application is started in a special mode known as Server mode or Embedded mode. This means that although the application is running and fully functional, there is no graphical user interface and nothing is visible on the screen. (You can, however, use Task Manager to verify that the process is running.) Server mode allows you to carry out actions without a user seeing, and possibly interfering with, those actions.

Although server mode is often useful in system administration scripting, sometimes you might want a user interface (for example, if you are displaying data in Internet Explorer). If so, you will need to use the appropriate command for that **COM** object to make the application appear on screen. For example, the following script creates an instance of Internet Explorer and then uses the Visible command to allow the user to see the application:

```
Set oIE = CreateObject("InternetExplorer.Application")  
oIE.Visible = True
```

Binding

Binding refers to the way that a script or an application accesses a **COM** object. When you create an object reference to an Automation object, **VBScript** must verify that the object exists and that any methods or properties you attempt to access are valid and are called correctly. This process of connecting to and

verifying an object and its methods and properties is known as binding.

COM supports two types of binding: early and late. With early binding, an object, its methods, and its properties are checked when the application is compiled. If there are any problems, compilation will fail. Early binding is faster than late binding because the object is verified before the application runs. In addition, early binding provides access to the objects type library, which contains information about the methods and properties of the object. The information in the type library can then be included within the compiled code and thus be available whenever the application needs it.

Because **VBScript** is not a compiled language, it does not support early binding. Instead, you must use late binding, in which binding does not occur until the script actually runs. With late binding, the script must access the registry to obtain information about the object, its methods, and its properties. Because **VBScript** does not have access to the objects type library, it must perform a similar lookup any time it accesses the object or attempts to use one of the objects methods or properties. In addition, any incorrect calls to the object will not be found until the script actually runs.

Choosing a Method for Binding to an Automation Object

Binding to an **Automation object** is actually quite easy; the hardest part involves knowing how to bind to that object (that is, do you use the **GetObject** method or the **CreateObject** method?). For the most part, this depends on the object you are binding to; however, some general guidelines for binding to Automation objects are listed in the following table

To Perform This Task	Use This Method
Bind to WMI or ADSI.	VBScript GetObject and the appropriate moniker. A moniker is an intermediate object that makes it possible to locate, activate, and create a reference to Automation objects. Both WMI and ADSI are accessed using monikers; this allows your script to locate WMI and ADSI objects without having to know the physical location of these objects. Monikers are typically used to bind to COM objects that reside outside the file system.
Bind to a new instance of an Automation object.	VBScript CreateObject and the appropriate ProgID. Set oTestObject = CreateObject("Word.Application")
Bind to an existing instance of an Automation object.	VBScript GetObject and the appropriate ProgID. Set oTestObject = GetObject("Word.Application")
Bind to an Automation object by using an existing file.	VBScript GetObject and the appropriate file path. Set oTestObject = GetObject("c:\scripts\test.xls")
Bind to a new instance of an Automation object, with the ability to receive event notifications from that object.	Wscript CreateObject , the appropriate ProgID, and an event mapping variable. Set oTestObject = Wscript.CreateObject _ ("Word.Application", "Word") NOT SUPPORTED BY QUICKTEST
Bind to an existing instance of an Automation object, with the ability to receive event notifications from that object.	Wscript GetObject , the appropriate ProgID, and an event mapping variable. Set oTestObject = Wscript.GetObject _ ("Word.Application", "Word") NOT SUPPORTED BY QUICKTEST
Bind to a new instance of an Automation object on a remote computer.	VBScript CreateObject , the appropriate

Verifying Object References

The **IsObject** function allows you to verify that you were able to obtain an object reference. If the **GetObject** or **CreateObject** call succeeds, **IsObject** will return **True** (-1). If the **GetObject** or **CreateObject** call fails, **IsObject** will return **False** (0). For example, the following code uses **CreateObject** to try to obtain an object reference (assigned to the variable *oTestObject*) to a nonexistent object. Because the object call fails, **TestObject** is not assigned an object reference, and **IsObject** returns 0.

```
On Error Resume Next
Set oTestObject = CreateObject("Fake.Object")
MsgBox IsObject(oTestObject)
```

Unfortunately, **VBScript** assumes that once an object reference has been established, that reference will remain valid for the lifetime of the script. That is generally not a problem, particularly for **ADSI** and **WMI**, which are unlikely to disappear while the script is running.

The same cannot be said for other **Automation objects**, however. For example, consider the following script, which starts an instance of **Microsoft Word**, immediately stops that instance, and then uses **IsObject** to test whether the object reference is still valid:

```
Set oTestObject = CreateObject("Word.Application")
oTestObject.Quit
MsgBox IsObject(oTestObject)
```

When the script runs, **IsObject** reports that *oTestObject* is still an object because *oTestObject* is still an object reference; it just no longer points to a running instance of **Microsoft Word**.

There are two ways to work around this problem. One approach is to use **WMI** to verify that the process (in this case, *Winword.exe*) is still running. Although this method will work, it requires you to repeatedly query the set of running processes on the computer, something that will slow your script. In addition, matters can get complicated if multiple instances of *Winword.exe* are running because there is no straightforward method for identifying the instance of *Winword.exe* that you created and that your object reference refers to. To avoid possible problems (such as a script that inadvertently deletes text from the wrong Word document), your script should use the same instance of *Winword.exe* all the way through.

A better approach is to use the **Is Nothing** statements on an Automation object.

Unloading Objects from Memory

In-process servers (that is, Automation objects encapsulated in .dll files) will automatically unload themselves from memory when the calling script completes. This is because these objects run in the same process as the script; when the script process ends and is thus removed from memory, any in-process servers will also be stopped and removed from memory. For example, the following script creates an instance of the **FileSystemObject** and then displays a message box. As soon as you dismiss the message box, both the script and the **FileSystemObject** are removed from memory.

```
Set oTestObject = CreateObject("Scripting.FileSystemObject")
ExitTest(0)
```

This is not true, however, for out-of-process servers, Automation objects that run in a different process than the script itself. For example, the following script creates an instance of **Microsoft Word** and then displays a message box. When you dismiss the message box, the script process is unloaded from memory.

```
Set oTestObject = CreateObject("Word.Application")
ExitTest(0)
```

However, the **Microsoft Word** process (*Winword.exe*) will continue to run and remain in memory, even though it is not visible on the screen. This is because there is no inherent tie between the script process and the Word process; anything you do to the script process does not affect the Word process and vice versa. You can verify that the process is still running and verify the amount of memory it is still allocated by using Task Manager

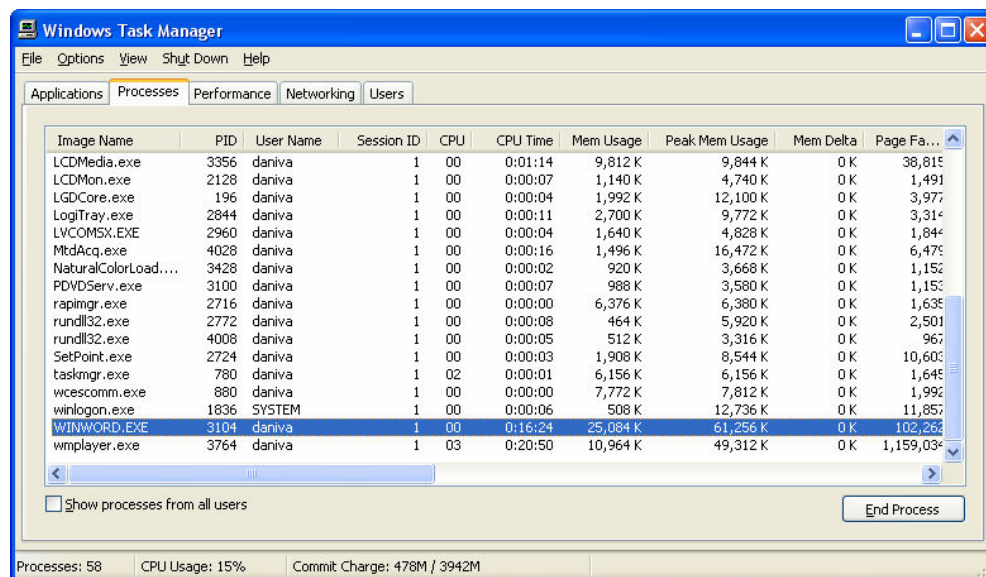


Figure 5 - Automation Object Running After a Script Has Completed

With out-of-process servers, you will typically have to use the method built into the object to explicitly unload it from memory. (You will need to check the documentation for the object to determine that method.) **Microsoft Word**, for example, is unloaded from memory by using the **Quit** method. The following script creates an instance of **Microsoft Word** and then immediately unloads that instance using the **Quit** method.

```
Set oTestObject = CreateObject("Word.Application")
oTestObject.Quit
```

If you run the preceding script and then check the processes running on the computer, you will not see Winword.exe (unless, of course, you had multiple copies of Winword.exe running).

Nothing Keyword

VBScript includes the Nothing keyword, which can be used to disassociate an object reference and an object. After an object variable is set to **Nothing**, the

variable no longer maintains an object reference and thus cannot be used to control the object. For example, the following code creates an instance of **Microsoft Word**, sets the object variable to *oTestObject*, and then tries to use *oTestObject* to quit **Word** and unload the object from memory.

```
Set oTestObject = CreateObject("Word.Application")
Set oTestObject = Nothing
oTestObject.Quit
```

When this script runs, the error message shown in Figure 6 appears. The script fails because *oTestObject* no longer represents a valid reference.

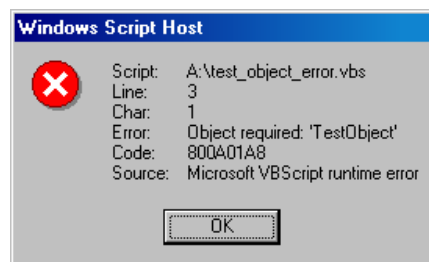


Figure 6 - Working with an Invalid Object Reference

Setting an object variable to **Nothing** releases a small amount of memory but does not unload the object itself from memory. Because of that, there is generally no reason to set an object variable to **Nothing**; in effect, object variables (and all other variables, for that matter) are set to **Nothing** when the script completes. For example, in the following script the last line of code is superfluous: It sets the object variable *oTestVariable* to **Nothing**, but that would occur anyway as soon as the script ended.

```
Set oTestObject = CreateObject("Scripting.FileSystemObject")
Set oTestObject = Nothing
```

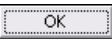
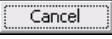

Q&A

Desktop Management

How To List Items in the Administrative Tools Folder?

Appendix 4.A

MsgBox Return Value Constants

Constant	Value	Description
vbOK	1	OK 
vbCancel	2	Cancel 
vbAbort	3	Abort 

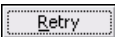

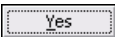
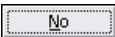
vbRetry	4	Retry	
vbIgnore	5	Ignore	
vbYes	6	Yes	
vbNo	7	No	

Table 1 - MsgBox Return Values**Button Display Constants**

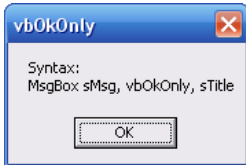
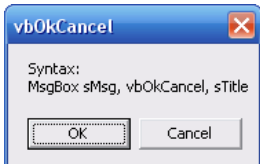
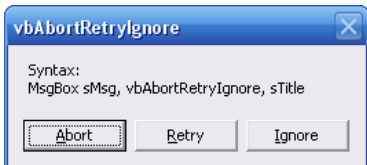
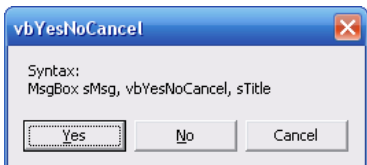
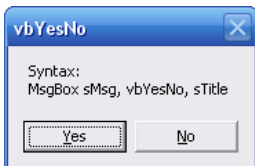
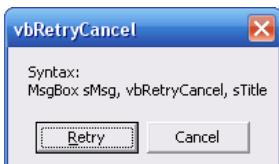
Constant	Value	Display
vbOKOnly	1	
vbOkCancel	2	
vbAbortRetryIgnore	3	
vbYesNoCancel	4	
vbYesNo	5	
vbRetryCancel	7	

Table 2 - Button Display Constants**Icon Display Constants**

Constant	Value	Display
----------	-------	---------





vbCritical	16	
vbQuestion	32	
vbExclamation	48	
vbInformation	64	

Table 3 - Icon Display Constants

Default Button Constants

Constant	Value	Default button
vbDefaultButton1	0	First button
vbDefaultButton2	256	Second button
vbDefaultButton3	512	Third button
vbDefaultButton4	768	Fourth button

Table 4 – Default Button Constants

Modality Constants

Constant	Value	Modality
vbApplicationModal	0	Application
vbSystemModal	4096	System

Table 5 – Modality Constants

Comparison Constants

Constant	Value	Description
vbBinaryCompare	0	Perform a binary comparison.
vbTextCompare	1	Perform a textual comparison.
VBDataBaseCompare	2	Compare information inside database

Table 6 – Comparison Constants

Interval Settings

Setting	Description
yyyy	Year
q	Quarter
m	Month
y	Day of Year
d	Day
w	Weekday
ww	Week of Year

h	Hour
n	Minute
s	Second

Table 7 – Interval Settings

Tristate Constants

Constant	Value	Default button
TristateTrue	-1	True
TristateFalse	0	False
TristateUseDefault	-2	Use the setting from the computer's regional settings.

Table 8 – Tristate Constants

Color Constants

Constant	Value	Color
VBBlack	&h00	Black
VBRed	&hFF	Red
VBGreen	&hFF00	Green
VBYellow	&hFFFF	Yellow
VBBlue	&hFF0000	Blue
VBMagenta	&hFF00FF	Magenta
VCyan	&hFFFF00	Cyan
VBWhite	&hFFFFFF	White

Table 9 – Color Constants

Date Format Constants

Constant	Value	Description
VBGeneralDate	0	Display the date and time using system settings
VBLongDate	1	Display the date in long date format (June 26, 1943)
VBShortDate	2	Display the date in short date format (6/26/43)
VBLongTime	3	Display the time in long time format (3:48:01 PM)
VBShortTime	4	Display the time in short time format (24 hour clock) (15:48)

Table 10 – Date Format Constants

Date and Time Constants

Constant	Value	Description
VBSunday	1	Sunday
VBMonday	2	Monday
VBTuesday	3	Tuesday
VBWednesday	4	Wednesday
VBThursday	5	Thursday

VBFriday	6	Friday
VBSaturday	7	Saturday
VBFirstJan1	1	Week of January 1
VBFirstFourDays	2	First week of the year that has at least four days
VBFirstFullWeek	3	First full week of the year

Table 11 – Date and Time Constants

String Constants

Constant	Value	Description
VBCR	Chr(13)	Carriage return
VBCrLf	Chr(13) Chr(10)	Combined carriage return and line feed
VBFormFeed	Chr(12)	Form feed
VBLF	Chr(10)	Line feed
VBNewLine	Chr(13) Chr(10) Or Chr(10)	Newline character appropriate for platform
VBNullChar	Chr(0)	Character value of zero
VBNullString	String Of Value Zero	Null string
VBTab	Chr(9)	Horizontal (row) tab
VBVerticalTab	Chr(11)	Vertical (column) tab

Table 12 – String Constants

VarType Constants

Constant	Value	Description
VBEmpty	0	Uninitialized
VBNull	1	Contains no valid data
VBInteger	2	Integer subtype
VBLong	3	Long subtype
VBSingle	4	Single subtype
VBDouble	5	Double subtype
VBCurrency	6	Currency subtype
VBDate	7	Date subtype
VBString	8	String subtype
VBObject	9	Object
VBError	10	Error subtype
VBBoolean	11	Boolean subtype
VBVariant	12	Variant (only use for arrays of variants)
VBDataObject	13	Data access object
VBDecimal	14	Decimal subtype
VBByte	17	Byte subtype
VBArray	8192	Array

Table 13 – Vertype Constants

WeekDay Constants

Parameter	Value	Description
<i>vbSunday</i>	1	Sunday
<i>vbMonday</i>	2	Monday
<i>vbTuesday</i>	3	Tuesday
<i>vbWednesday</i>	4	Wednesday
<i>vbThursday</i>	5	Thursday
<i>vbFriday</i>	6	Friday
<i>vbSaturday</i>	7	Saturday

Table 14 – WeekDay Constants

Appendix 4.B

Locale Id's

The following table lists the **locale IDs** used by the **GetLocale** and **SetLocale** functions. The **GetLocale** function returns a **Long** containing the decimal **locale ID**. In most cases, the **SetLocale** function accepts a **locale ID** in the form of a decimal, a hexadecimal, or a string value.

Locale	Decimal ID	Hex ID	String ID
Afrikaans	1078	&h0436	af
Albanian	1052	&h041C	sq
Arabic (No location)	1	&h0001	ar
Arabic (United Arab Emirates)	14337	&h3801	ar-ae
Arabic (Bahrain)	15361	&h3C01	ar-bh
Arabic (Algeria)	5121	&h1401	ar-dz
Arabic (Egypt)	3073	&h0C01	ar-eg
Arabic (Iraq)	2049	&h0801	ar-iq
Arabic (Jordan)	11265	&h2C01	ar-jo
Arabic (Kuwait)	13313	&h3401	ar-kw
Arabic (Lebanon)	12289	&h3001	ar-lb
Arabic (Libya)	4097	&h1001	ar-ly
Arabic (Morocco)	6145	&h1801	ar-ma
Arabic (Oman)	8193	&h2001	ar-om
Arabic (Qatar)	16385	&h4001	ar-qa
Arabic (Saudi Arabia)	1025	&h0401	ar-sa
Arabic (Syria)	10241	&h2801	ar-sy
Arabic (Tunisia)	7169	&h1C01	ar-tn
Arabic (Yemen)	9217	&h2401	ar-ye

Azeri (Latin)	1068	&h042C	az-az
Basque	1069	&h042D	eu
Belarusian	1059	&h0423	be
Bulgarian	1026	&h0402	bg
Catalan	1027	&h0403	ca
Chinese (No location)	4	&h0004	zh
Chinese (China)	2052	&h0804	zh-cn
Chinese (Hong Kong S.A.R.)	3076	&h0C04	zh-hk
Arabic (United Arab Emirates)	14337	&h3801	ar-ae
Arabic (Bahrain)	15361	&h3C01	ar-bh
Arabic (Algeria)	5121	&h1401	ar-dz
Arabic (Egypt)	3073	&h0C01	ar-eg
Arabic (Iraq)	2049	&h0801	ar-iq
Arabic (Jordan)	11265	&h2C01	ar-jo
Arabic (Kuwait)	13313	&h3401	ar-kw
Arabic (Lebanon)	12289	&h3001	ar-lb
Arabic (Libya)	4097	&h1001	ar-ly
Arabic (Morocco)	6145	&h1801	ar-ma
Chinese (Singapore)	4100	&h1004	zh-sg
Chinese (Taiwan)	1028	&h0404	zh-tw
Croatian	1050	&h041A	hr
Czech	1029	&h0405	cs
Danish	1030	&h0406	da
Dutch (The Netherlands)	1043	&h0413	nl
Dutch (Belgium)	2067	&h0813	nl-be
English (No location)	9	&h0009	en
English (Australia)	3081	&h0C09	en-au
English (Belize)	10249	&h2809	en-bz
English (Canada)	4105	&h1009	en-ca
English (Caribbean)	9225	&h2409	
English (Ireland)	6153	&h1809	en-ie
English (Jamaica)	8201	&h2009	en-jm
English (New Zealand)	5129	&h1409	en-nz
English (Philippines)	13321	&h3409	en-ph
English (South Africa)	7177	&h1C09	en-za
English (Trinidad)	11273	&h2C09	en-tt
English (United Kingdom)	2057	&h0809	en-gb
English (United States)	1033	&h0409	en-us
Estonian	1061	&h0425	et

Farsi	1065	&h0429	fa
Finnish	1035	&h040B	fi
Faroese	1080	&h0438	fo
French (France)	1036	&h040C	fr
French (Belgium)	2060	&h080C	fr-be
French (Canada)	3084	&h0C0C	fr-ca
French (Luxembourg)	5132	&h140C	fr-lu
French (Switzerland)	4108	&h100C	fr-ch
Gaelic (Ireland)	2108	&h083C	
Gaelic (Scotland)	1084	&h043C	gd
German (Germany)	1031	&h0407	de
German (Austria)	3079	&h0C07	de-at
German (Liechtenstein)	5127	&h1407	de-li
German (Luxembourg)	4103	&h1007	de-lu
German (Switzerland)	2055	&h0807	de-ch
Greek	1032	&h0408	el
Hebrew	1037	&h040D	he
Hindi	1081	&h0439	hi
Hungarian	1038	&h040E	hu
Icelandic	1039	&h040F	is
Indonesian	1057	&h0421	in
Italian (Italy)	1040	&h0410	it
Italian (Switzerland)	2064	&h0810	it-ch
Japanese	1041	&h0411	ja
Korean	1042	&h0412	ko
Latvian	1062	&h0426	lv
Lithuanian	1063	&h0427	lt
FYRO Macedonian	1071	&h042F	mk
Malay (Malaysia)	1086	&h043E	ms
Maltese	1082	&h043A	mt
Marathi	1102	&h044E	mr
Norwegian (Bokmål)	1044	&h0414	no
Norwegian (Nynorsk)	2068	&h0814	
Polish	1045	&h0415	pl
Portuguese (Portugal)	2070	&h0816	pt
Portuguese (Brazil)	1046	&h0416	pt-br
Raeto-Romance	1047	&h0417	rm
Romanian (Romania)	1048	&h0418	ro
Romanian (Moldova)	2072	&h0818	ro-mo

Russian	1049	&h0419	ru
Russian (Moldova)	2073	&h0819	ru-mo
Sanskrit	1103	&h044F	
Serbian (Cyrillic)	3098	&h0C1A	sr
Serbian (Latin)	2074	&h081A	
Setsuana	1074	&h0432	tn
Slovenian	1060	&h0424	sl
Slovak	1051	&h041B	sk
Sorbian	1070	&h042E	sb
Spanish (Spain)	1034	&h0C0A	es
Spanish (Argentina)	11274	&h2C0A	es-ar
Spanish (Bolivia)	16394	&h400A	es-bo
Spanish (Chile)	13322	&h340A	es-cl
Spanish (Colombia)	9226	&h240A	es-co
Spanish (Costa Rica)	5130	&h140A	es-cr
Spanish (Dominican Republic)	7178	&h1C0A	es-do
Spanish (Ecuador)	12298	&h300A	es-ec
Spanish (Guatemala)	4106	&h100A	es-gt
Spanish (Honduras)	18442	&h480A	es-hn
Spanish (Mexico)	2058	&h080A	es-mx
Spanish (Nicaragua)	19466	&h4C0A	es-ni
Spanish (Panama)	6154	&h180A	es-pa
Spanish (Peru)	10250	&h280A	es-pe
Spanish (Puerto Rico)	20490	&h500A	es-pr
Spanish (Paraguay)	15370	&h3C0A	es-py
Spanish (El Salvador)	17418	&h440A	es-sv
Spanish (Uruguay)	14346	&h380A	es-uy
Spanish (Venezuela)	8202	&h200A	es-ve
Sutu	1072	&h0430	sx
Swahili	1089	&h0441	
Swedish (Sweden)	1053	&h041D	sv
Swedish (Finland)	2077	&h081D	sv-fi
Tamil	1097	&h0449	
Tatar	1092	0X0444	
Thai	1054	&h041E	th
Turkish	1055	&h041F	tr
Tsonga	1073	&h0431	ts
Ukrainian	1058	&h0422	uk
Urdu	1056	&h0420	ur

Uzbek (Cyrillic)	2115	&h0843	uz-uz
Uzbek (Latin)	1091	&h0443	uz-uz
Vietnamese	1066	&h042A	vi
Xhosa	1076	&h0434	xh
Yiddish	1085	&h043D	
Zulu	1077	&h0435	zu
Sutu	1072	&h0430	sx
Swahili	1089	&h0441	
Swedish (Sweden)	1053	&h041D	sv
Swedish (Finland)	2077	&h081D	sv-fi
Tamil	1097	&h0449	
Tatar	1092	0X0444	
Thai	1054	&h041E	th
Turkish	1055	&h041F	tr
Tsonga	1073	&h0431	ts
Ukrainian	1058	&h0422	uk
Urdu	1056	&h0420	ur
Uzbek (Cyrillic)	2115	&h0843	uz-uz
Uzbek (Latin)	1091	&h0443	uz-uz
Vietnamese	1066	&h042A	vi
Xhosa	1076	&h0434	xh
Yiddish	1085	&h043D	
Zulu	1077	&h0435	zu
Swedish (Sweden)	1053	&h041D	sv
Swedish (Finland)	2077	&h081D	sv-fi
Tamil	1097	&h0449	
Tatar	1092	0X0444	
Thai	1054	&h041E	th
Turkish	1055	&h041F	tr
Tsonga	1073	&h0431	ts
Ukrainian	1058	&h0422	uk
Urdu	1056	&h0420	ur
Uzbek (Cyrillic)	2115	&h0843	uz-uz
Uzbek (Latin)	1091	&h0443	uz-uz
Vietnamese	1066	&h042A	vi
Xhosa	1076	&h0434	xh
Yiddish	1085	&h043D	
Zulu	1077	&h0435	zu
Swedish (Sweden)	1053	&h041D	sv

Swedish (Finland)	2077	&h081D	sv-fi
Tamil	1097	&h0449	
Tatar	1092	0X0444	
Thai	1054	&h041E	th
Turkish	1055	&h041F	tr
Tsonga	1073	&h0431	ts
Ukrainian	1058	&h0422	uk
Urdu	1056	&h0420	ur
Uzbek (Cyrillic)	2115	&h0843	uz-uz
Uzbek (Latin)	1091	&h0443	uz-uz
Vietnamese	1066	&h042A	vi
Xhosa	1076	&h0434	xh
Yiddish	1085	&h043D	
Zulu	1077	&h0435	zu

Table 15 – Locale ID's