Visual Basic Scripting Edition

VBScript

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

VBScript User's Guide

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

What Is VBScript?

Microsoft Visual Basic Scripting Edition brings active scripting to a wide variety of environments, including Web client scripting in Microsoft Internet Explorer and Web server scripting in Microsoft Internet Information Service.

Easy to Use and Learn

If you already know Visual Basic or Visual Basic for Applications (VBA), VBScript will be very familiar. Even if you do not know Visual Basic, once you learn VBScript, you are on your way to programming with the whole family of Visual Basic languages. Although you can learn about VBScript in just these few Web pages, they do not teach you how to program. To learn programming, take a look at *Step by Step* books available from Microsoft Press.

Windows Script

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.

VBScript in Other Applications and Browsers

As a developer, you can license VBScript source implementation at no charge for use in your products. Microsoft provides binary implementations of VBScript for the 32-bit Windows® API, the 16-bit Windows API, and the Macintosh®. VBScript is integrated with World Wide Web browsers. VBScript and Windows Script can also be used as a general scripting language in other applications.

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Adding VBScript Code to an HTML Page

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

The <SCRIPT> Tag

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

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

Beginning and ending <SCRIPT> tags surround the code. The LANGUAGE attribute indicates the scripting language. You must specify the language because browsers can use other scripting languages. Notice that the `CanDeliver` function is embedded in comment tags (<!— and —>). This prevents browsers that don't understand the <SCRIPT> tag from displaying the code.

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

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

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

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

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

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

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

VBScript Features not in Visual Basic for Applications

The following table lists VBScript features not in Visual Basic for Applications.

Category	Feature/Keyword
Declarations	Class
Miscellaneous	Eval Execute
Objects	RegExp
Script Engine Identification	ScriptEngine ScriptEngineBuildVersion ScriptEngineMajorVersion ScriptEngineMinorVersion

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Visual Basic for Applications Features Not In VBScript

The following table lists Visual Basic for Applications Features not in VBScript.

Category	Omitted Feature/Keyword
Array Handling	Option Base

	Declaring arrays with lower bound <> 0
Collection	Add, Count, Item, Remove Access to collections using ! character
Conditional Compilation	#Const #If...Then...#Else
Control Flow	DoEvents GoSub...Return, GoTo On Error GoTo On...GoSub, On...GoTo Line numbers, Line labels
Conversion	CVar, CVDate Str, Val
Data Types	All intrinsic data types except Variant Type...End Type
Date/Time	Date statement, Time statement
DDE	LinkExecute, LinkPoke, LinkRequest, LinkSend
Debugging	Debug.Print End, Stop
Declaration	Declare (for declaring DLLs) Optional ParamArray Static
Error Handling	Erl Error Resume, Resume Next
File Input/Output	All traditional Basic file I/O
Financial	All financial functions
Object Manipulation	TypeOf
Objects	Clipboard Collection
Operators	Like
Options	Def*type* Option Base Option Compare Option Private Module

Select Case	Expressions containing **Is** keyword or any comparison operators Expressions containing a range of values using the **To** keyword.
Strings	Fixed-length strings LSet, RSet Mid Statement StrConv
Using Objects	Collection access using !

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

VBScript Fundamentals

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

A Simple VBScript Page

A Simple Page

With Microsoft® Internet Explorer, you can view the page produced by the following HTML code. If you click the button on the page, you see VBScript in action.

```
<HTML>
<HEAD><TITLE>A Simple First Page</TITLE>
<SCRIPT LANGUAGE="VBScript">
<!--
Sub Button1_OnClick
   MsgBox "Mirabile visu."
End Sub
-->
</SCRIPT>
</HEAD>
<BODY>
<H3>A Simple First Page</H3><HR>
<FORM><INPUT NAME="Button1" TYPE="BUTTON" VALUE="Click Here"></FORM>
</BODY>
</HTML>
```

The result is a little underwhelming: a dialog box displays a Latin phrase ("Wonderful to behold"). However, there's quite a bit going on.

When Internet Explorer reads the page, it finds the <SCRIPT> tags, recognizes there is a piece of VBScript code, and saves the code. When you click the button, Internet Explorer makes the connection between the button and the code, and runs the procedure.

The **Sub** procedure in the <SCRIPT> tags is an event procedure. There are two parts to the procedure name: the name of the button, `Button1` (from the NAME attribute in the <INPUT> tag), and an event name, `OnClick`. The two names are joined by an underscore(_). Any time the button is clicked, Internet Explorer looks for and runs the corresponding event procedure, `Button1_OnClick`.

Internet Explorer defines the events available for form controls in the Internet Explorer Scripting Object Model documentation, which can be found on the Microsoft® Web site (http://www.microsoft.com).

Pages can use combinations of controls and procedures, too. VBScript and Forms shows some simple interactions between controls.

Other Ways to Attach Code to Events

Although the preceding way is probably the simplest and most general, you can attach VBScript code to events in two other ways. Internet Explorer allows you to add short sections of inline code in the tag defining the control. For example, the following <INPUT> tag performs the same action as the previous code example when you click the button:

```
<INPUT NAME="Button1" TYPE="BUTTON"
   VALUE="Click Here" OnClick='MsgBox "Mirabile visu."'>
```

Notice that the function call itself is enclosed in single quotation marks, and the string for the **MsgBox** function is enclosed in double quotation marks. You can use multiple statements as long as you separate the statements with colons (:).

You can also write a <SCRIPT> tag so that it applies only to a particular event for a specific control:

```
<SCRIPT LANGUAGE="VBScript" EVENT="OnClick" FOR="Button1">
<!--
   MsgBox "Mirabile visu."
-->
</SCRIPT>
```

Because the <SCRIPT> tag already specifies the event and the control, you don't use **Sub** and **End Sub** statements.

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

VBScript Features

The following table is a list of VBScript features.

Category	Keywords
Array handling	Array Dim, Private, Public, ReDim IsArray Erase LBound, UBound
Assignments	Set
Comments	Comments using ' or Rem
Constants/Literals	Empty Nothing Null True, False
Control flow	Do...Loop For...Next For Each...Next If...Then...Else Select Case While...Wend With
Conversions	Abs Asc, AscB, AscW Chr, ChrB, ChrW CBool, CByte CCur, CDate CDbl, CInt CLng, CSng, CStr DateSerial, DateValue Hex, Oct Fix, Int Sgn TimeSerial, TimeValue

Dates/Times	Date, Time DateAdd, DateDiff, DatePart DateSerial, DateValue Day, Month, MonthName Weekday, WeekdayName, Year Hour, Minute, Second Now TimeSerial, TimeValue
Declarations	Class Const Dim, Private, Public, ReDim Function, Sub Property Get, Property Let, Property Set
Error Handling	On Error Err
Expressions	Eval Execute RegExp Replace Test
Formatting Strings	FormatCurrency FormatDateTime FormatNumber FormatPercent
Input/Output	InputBox LoadPicture MsgBox
Literals	Empty False Nothing Null True
Math	Atn, Cos, Sin, Tan Exp, Log, Sqr Randomize, Rnd
Miscellaneous	Eval Function

	Execute Statement
	RGB Function
Objects	CreateObject
	Err Object
	GetObject
	RegExp
Operators	Addition (+), Subtraction (-)
	Exponentiation (^)
	Modulus arithmetic (Mod)
	Multiplication (*), Division (/)
	Integer Division (\)
	Negation (-)
	String concatenation (&)
	Equality (=), Inequality (<>)
	Less Than (<), Less Than or Equal To (<=)
	Greater Than (>)
	Greater Than or Equal To (>=)
	Is
	And, Or, Xor
	Eqv, Imp
Options	Option Explicit
Procedures	Call
	Function, Sub
	Property Get, Property Let, Property Set
Rounding	Abs
	Int, Fix, Round
	Sgn
Script Engine ID	ScriptEngine
	ScriptEngineBuildVersion
	ScriptEngineMajorVersion
	ScriptEngineMinorVersion
Strings	Asc, AscB, AscW
	Chr, ChrB, ChrW
	Filter, InStr, InStrB
	InStrRev
	Join
	Len, LenB

	LCase, UCase Left, LeftB Mid, MidB Right, RightB Replace Space Split StrComp String StrReverse LTrim, RTrim, Trim
Variants	IsArray IsDate IsEmpty IsNull IsNumeric IsObject TypeName VarType

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

VBScript Data Types

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

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

Variant Subtypes

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

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

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

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

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

VBScript Variables

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

Declaring Variables

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

```
Dim DegreesFahrenheit
```

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

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

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

Naming Restrictions

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

- Must begin with an alphabetic character.
- Cannot contain an embedded period.
- Must not exceed 255 characters.

- Must be unique in the scope in which it is declared.

Scope and Lifetime of Variables

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

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

Assigning Values to Variables

Values are assigned to variables creating an expression as follows: the variable is on the left side of the expression and the value you want to assign to the variable is on the right. For example:

```
B = 200
```

Scalar Variables and Array Variables

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

```
Dim A(10)
```

Although the number shown in the parentheses is 10, all arrays in VBScript are zero-based, so this array actually contains 11 elements. In a zero-based array, the number of array elements is always the number shown in parentheses plus one. This kind of array is called a fixed-size array.

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

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

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

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

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

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

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

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

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

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

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

There is no limit to the number of times you can resize a dynamic array, although if you make an array smaller, you lose the data in the eliminated elements.

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

VBScript Constants

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

Creating Constants

You create user-defined constants in VBScript using the Const statement. Using the **Const** statement, you can create string or numeric constants with meaningful names and assign them literal values. For example:

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

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

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

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

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

VBScript Operators

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

Operator Precedence

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

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

Arithmetic

Description	Symbol
Exponentiation	^
Unary negation	-
Multiplication	*
Division	/
Integer division	\
Modulus arithmetic	Mod
Addition	+
Subtraction	-
String concatenation	&

Comparison

Description	Symbol

Equality	=
Inequality	<>
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=
Object equivalence	Is

Logical

Description	Symbol
Logical negation	Not
Logical conjunction	And
Logical disjunction	Or
Logical exclusion	Xor
Logical equivalence	Eqv
Logical implication	Imp

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

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

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Using Conditional Statements

Controlling Program Execution

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

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

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

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

Running Statements if a Condition is True

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

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

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

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

```
End Sub
```

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

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

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

Deciding Between Several Alternatives

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

```
Sub ReportValue(value)
   If value = 0 Then
      MsgBox value
   ElseIf value = 1 Then
      MsgBox value
   ElseIf value = 2 then
      Msgbox value
   Else
      Msgbox "Value out of range!"
   End If
```

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

Making Decisions with Select Case

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

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

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

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

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Looping Through Code

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

The following looping statements are available in VBScript:

- Do...Loop: Loops while or until a condition is **True**.
- While...Wend: Loops while a condition is **True**.
- For...Next: Uses a counter to run statements a specified number of times.
- For Each...Next: Repeats a group of statements for each item in a collection or each element of an array.

Using Do Loops

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

Repeating Statements While a Condition is True

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

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

Sub ChkLastWhile()
   Dim counter, myNum
   counter = 0
   myNum = 9
   Do
      myNum = myNum - 1
```

```
      counter = counter + 1
   Loop While myNum > 10
   MsgBox "The loop made " & counter & " repetitions."
End Sub
```

Repeating a Statement Until a Condition Becomes True

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

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

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

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

You can exit a **Do...Loop** by using the **Exit Do** statement. Because you usually want to exit only in certain situations, such as to avoid an endless loop, you should use the **Exit Do** statement in the **True** statement block of an **If...Then...Else** statement. If the condition is **False**, the loop runs as usual.

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

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

Using While...Wend

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

Using For...Next

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

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

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

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

```
Sub TwosTotal()
   Dim j, total
   For j = 2 To 10 Step 2
      total = total + j
```

```
   Next
   MsgBox "The total is " & total
End Sub
```

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

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

You can exit any **For...Next** statement before the counter reaches its end value by using the **Exit For** statement. Because you usually want to exit only in certain situations, such as when an error occurs, you should use the **Exit For** statement in the **True** statement block of an **If...Then...Else** statement. If the condition is **False**, the loop runs as usual.

Using For Each...Next

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

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

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

   For Each I in d
```

```
      Document.frmForm.Elements(I).Value = D.Item(I)
   Next
End Sub
-->
</SCRIPT>
<BODY>
<CENTER>
<FORM NAME="frmForm"

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

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

VBScript Procedures

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

Sub Procedures

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

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

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

Function Procedures

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

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

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

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

Getting Data into and out of Procedures

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

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

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

Using Sub and Function Procedures in Code

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

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

-or-

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

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

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

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

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

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

VBScript Coding Conventions

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

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

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

Constant Naming Conventions

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

```
USER_LIST_MAX
NEW_LINE
```

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

```
conYourOwnConstant
```

Variable Naming Conventions

To enhance readability and consistency, use the following prefixes with descriptive names for variables in your VBScript code.

Subtype	Prefix	Example
Boolean	bln	blnFound
Byte	byt	bytRasterData

Date (Time)	dtm	dtmStart
Double	dbl	dblTolerance
Error	err	errOrderNum
Integer	int	intQuantity
Long	lng	lngDistance
Object	obj	objCurrent
Single	sng	sngAverage
String	str	strFirstName

Variable Scope

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

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

Variable Scope Prefixes

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

Scope	Prefix	Example
Procedure-level	None	dblVelocity
Script-level	s	sblnCalcInProgress

Descriptive Variable and Procedure Names

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

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

Object Naming Conventions

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

Object type	Prefix	Example
3D Panel	pnl	pnlGroup
Animated button	ani	aniMailBox
Check box	chk	chkReadOnly
Combo box, drop-down list box	cbo	cboEnglish
Command button	cmd	cmdExit
Common dialog	dlg	dlgFileOpen
Frame	fra	fraLanguage
Horizontal scroll bar	hsb	hsbVolume
Image	img	imgIcon
Label	lbl	lblHelpMessage
Line	lin	linVertical
List Box	lst	lstPolicyCodes
Spin	spn	spnPages
Text box	txt	txtLastName
Vertical scroll bar	vsb	vsbRate
Slider	sld	sldScale

Code Commenting Conventions

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

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

Procedure header comments should include the following section headings. For examples, see the "Formatting Your Code" section that follows.

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

Remember the following points:

- Every important variable declaration should include an inline comment describing the use of the variable being declared.
- Variables, controls, and procedures should be named clearly to ensure that inline comments are only needed for complex implementation details.
- At the beginning of your script, you should include an overview that describes the script, enumerating objects, procedures, algorithms, dialog boxes, and other system dependencies. Sometimes a piece of pseudocode describing the algorithm can be helpful.

Formatting Your Code

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

- Indent standard nested blocks four spaces.
- Indent the overview comments of a procedure one space.
- Indent the highest level statements that follow the overview comments four spaces, with each nested block indented an additional four spaces.

The following code adheres to VBScript coding conventions.

```
'*********************************************************
' Purpose: Locates the first occurrence of a specified user
'          in the UserList array.
' Inputs: strUserList(): the list of users to be searched.
'         strTargetUser: the name of the user to search for.
' Returns: The index of the first occurrence of the strTargetUser
```

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

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

VBScript and Forms

Simple Validation

You can use Visual Basic Scripting Edition to do much of the form processing that you'd usually have to do on a server. You can also do things that just can't be done on the server.

Here's an example of simple client-side validation. The HTML code is for a text box and a button. If you use Microsoft® Internet Explorer to view the page produced by the following code, you'll see a small text box with a button next to it.

```
<HTML>
<HEAD><TITLE>Simple Validation</TITLE>
```

```
<SCRIPT LANGUAGE="VBScript">
<!--
Sub Validate
  Dim TheForm
  Set TheForm = Document.forms("ValidForm")
  If IsNumeric(TheForm.Text1.Value) Then
    If TheForm.Text1.Value < 1 Or TheForm.Text1.Value > 10 Then
      MsgBox "Please enter a number between 1 and 10."
    Else
      MsgBox "Thank you."
    End If
  Else
    MsgBox "Please enter a numeric value."
  End If
End Sub-->
</SCRIPT>
</HEAD>
<BODY>
<H3>Simple Validation</H3><HR>
<form id="ValidForm" action="nothing.asp" onsubmit="Validate(); return false;" language="jscript">
Enter a value between 1 and 10:
<input name="Text1" TYPE="TEXT" SIZE="2">
<input name="Submit" TYPE="Submit" VALUE="Submit">
</form>
</BODY>
</HTML>
```

The difference between this text box and the examples on A Simple VBScript Page is that the **Value** property of the text box is used to check the entered value. To get the **Value** property, the code has to qualify the reference to the name of the text box.

You can always write out the full reference `Document.ValidForm.Text1`. However, where you have multiple references to form controls, you'll want to do what was done here. First declare a variable. Then use the Set statement to assign the form to the variable `TheForm`. A regular assignment statement, such as Dim, doesn't work here; you must use **Set** to preserve the reference to an object.

Using Numeric Values

Notice that the example directly tests the value against a number: it uses the IsNumeric function to make sure the string in the text box is a number. Although VBScript automatically converts strings and numbers, it's always a good practice to test a user-entered value for its data subtype and to use conversion functions as necessary. When doing addition with text box values, convert the values explicitly to numbers because the plus sign (+) operator represents both addition and string concatenation. For example, if `Text1` contains "1" and `Text2` contains "2", you see the following results:

```
A = Text1.Value + Text2.Value   ' A is "12"
A = CDbl(Text1.Value) + Text2.Value   ' A is 3
```

Validating and Passing Data Back to the Server

The simple validation example uses a plain button control. If a Submit control was used, the example would never see the data to check it — everything would go immediately to the server. Avoiding the Submit control lets you check the data, but it doesn't submit the data to the server. That requires an additional line of code:

```
<SCRIPT LANGUAGE="VBScript">
<!--
Sub Button1_OnClick
 Dim TheForm
 Set TheForm = Document.ValidForm
 If IsNumeric(TheForm.Text1.Value) Then
    If TheForm.Text1.Value < 1 Or TheForm.Text1.Value > 10 Then
      MsgBox "Please enter a number between 1 and 10."
    Else
      MsgBox "Thank you."
      TheForm.Submit   ' Data correct; send to server.
    End If
 Else
    MsgBox "Please enter a numeric value."
 End If
End Sub
-->
</SCRIPT>
```

To send the data to the server, the code invokes the **Submit** method on the form object when the data is correct. From there, the server handles the data just as it otherwise would — except that the data is correct before it gets there. Find complete information about the **Submit** method and other methods in the Internet Explorer Scripting Object Model documentation, which can be found on the Microsoft® Web site (http://www.microsoft.com).

So far, you've seen only the standard HTML <FORM> objects. Internet Explorer also lets you exploit the full power of ActiveX® controls (formerly called OLE controls) and Java objects.

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

VBScript in Internet Explorer

A Simple VBScript Page

VBScript and Forms

Using VBScript with Objects

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Using VBScript with Objects

Using Objects

Whether you use an ActiveX® control (formerly called an OLE control) or a Java object, Microsoft Visual Basic Scripting Edition and Microsoft® Internet Explorer handle it the same way. If you're using Internet Explorer and have installed the **Label** control, you can see the page produced by the following code.

You include an object using the <OBJECT> tags and set its initial property values using <PARAM> tags. If you're a Visual Basic programmer, you'll recognize that using the <PARAM> tags is just like setting initial properties for a control on a form. For example, the following set of <OBJECT> and <PARAM> tags adds the ActiveX Label control to a page:

```
<OBJECT
   classid="clsid:99B42120-6EC7-11CF-A6C7-00AA00A47DD2"
```

```
   id=lblActiveLbl
   width=250
   height=250
   align=left
   hspace=20
   vspace=0
>
<PARAM NAME="Angle" VALUE="90">
<PARAM NAME="Alignment" VALUE="4">
<PARAM NAME="BackStyle" VALUE="0">
<PARAM NAME="Caption" VALUE="A Simple Desultory Label">
<PARAM NAME="FontName" VALUE="Verdana, Arial, Helvetica">
<PARAM NAME="FontSize" VALUE="20">
<PARAM NAME="FontBold" VALUE="1">
<PARAM NAME="FrColor" VALUE="0">
</OBJECT>
```

You can get properties, set properties, and invoke methods just as with any of the form controls. The following code, for example, includes <FORM> controls you can use to manipulate two properties of the Label control:

```
<FORM NAME="LabelControls">
<INPUT TYPE="TEXT" NAME="txtNewText" SIZE=25>
<INPUT TYPE="BUTTON" NAME="cmdChangeIt" VALUE="Change Text">
<INPUT TYPE="BUTTON" NAME="cmdRotate" VALUE="Rotate Label">
</FORM>
```

With the form defined, an event procedure for the `cmdChangeIt` button changes the label text:

```
<SCRIPT LANGUAGE="VBScript">
<!--
Sub cmdChangeIt_onClick
   Dim TheForm
   Set TheForm = Document.LabelControls
   lblActiveLbl.Caption = TheForm.txtNewText.Value
End Sub
-->
</SCRIPT>
```

The code qualifies references to controls and values inside the forms just as in the Simple Validation example.

Several ActiveX controls are available for use with Internet Explorer. You can find complete information about the properties, methods, and events there, as well as the class identifiers (CLSID) for the controls on the Microsoft® Web site (http://www.microsoft.com). You can find more information about the <OBJECT> tag on the *Internet Explorer 4.0 Author's Guide and HTML Reference* page.

> **Note** Earlier releases of Internet Explorer required braces ({}) around the classid attribute and did not conform to the W3C specification. Using braces with the current release generates a "This page uses an outdated version of the <OBJECT> tag" message.

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Introduction to Regular Expressions

The information contained in these pages is intended to provide an introduction to regular expressions in general.

While an attempt has been made to make each topic stand on it's own, much of the information contained in these topics relies upon the understanding of a previously introduced feature or concept. Therefore, it's recommended that you peruse these topics sequentially for the best overall understanding of the material.

The Introduction to Regular Expressions consists of the following individuals topics:

Order of Precedence

Ordinary Characters

Special Characters

Non-Printable Characters

Character Matching

Quantifiers

Anchors

Alternation and Grouping

Backreferences

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Regular Expressions

Unless you have worked with regular expressions before, the term and the concept may be unfamiliar to you. However, they may not be as unfamiliar as you think.

Think about how you search for files on your hard disk. You most likely use the ? and * characters to help find the files you're looking for. The ? character matches a single character in a file name, while the * matches zero or more characters. A pattern such as 'data?.dat' would find the following files:

data1.dat

data2.dat

datax.dat

dataN.dat

Using the * character instead of the ? character expands the number of files found. 'data*.dat' matches all of the following:

data.dat

data1.dat

data2.dat

data12.dat

datax.dat

dataXYZ.dat

While this method of searching for files can certainly be useful, it is also very limited. The limited ability of the ? and * wildcard characters give you an idea of what regular expressions can do, but regular expressions are much more powerful and flexible.

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Early Beginnings

Regular expressions trace their ancestry back to early research on how the human nervous system works. Warren McCulloch and Walter Pitts, a pair of neuro-physiologists, developed a mathematical way of describing these neural networks.

In 1956, an mathematician named Stephen Kleene, building on the earlier work of McCulloch and Pitts, published a paper entitled, *Representation of Events in Nerve Nets* that introduced the concept of regular expressions. Regular expressions were expressions used to describe what he called "the algebra of regular sets". hence the term "regular expression."

Subsequently, his work found its way into some early efforts with computational search algorithms done by Ken Thompson, the principal inventor of Unix. The first practical application of regular expressions was in the Unix editor called *qed*.

And the rest, as they say, is history. Regular expressions have been an important part of text-based editors and search tools ever since.

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Uses for Regular Expressions

In a typical search and replace operation, you must provide the exact text you are looking for. That technique may be adequate for simple search and replace tasks in static text, but it lacks flexibility and makes searching dynamic text difficult, if not impossible.

With regular expressions, you can:

- Test for a pattern within a string. For example, you can test an input string to see if a telephone number pattern or a credit card number pattern occurs within the string. This is called data validation.
- Replace text. You can use a regular expression to identify specific text in a document and either remove it completely or replace it with other text.
- Extract a substring from a string based upon a pattern match. You can find specific text within a document or input field

For example, if you need to search an entire web site to remove some outdated material and replace some HTML formatting tags, you can use a regular expression to test each file to see if the material or the HTML formatting tags you are looking for exists in that file. That way, you can narrow down the affected files to only those that contain the material that has to be removed or changed. You can then use a regular expression to remove the outdated material, and finally, you can use regular expressions to search for and replace the tags that need replacing.

Another example of where a regular expression is useful occurs in a language that isn't known for its string-handling ability. VBScript, a subset of Visual Basic, has a rich set of string-handling functions. JScript, like C, does not. Regular expressions provide a significant improvement in string-handling for JScript. However, regular expressions may also be more efficient to use in VBScript as well, allowing you do perform multiple string manipulations in a single expression.

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Regular Expression Syntax

A regular expression is a pattern of text that consists of ordinary characters (for example, letters a through z) and special characters, known as *metacharacters*. The pattern describes one or more strings to match when searching a body of text. The regular expression serves as a template for matching a character pattern to the string being searched.

Here are some examples of regular expression you might encounter:

JScript	VBScript	Matches
/^\[\t]*$/	"^\[\t]*$"	Match a blank line.
/\d{2}-\d{5}/	"\d{2}-\d{5}"	Validate an ID number consisting of 2 digits, a hyphen, and another 5 digits.
/<(.*)>.*<\/\1>/	"<(.*)>.*<\/\1>"	Match an HTML tag.

The following table contains the complete list of metacharacters and their behavior in the context of regular expressions:

Character	Description
\	Marks the next character as either a special character, a literal, a backreference, or an octal escape. For example, 'n'

<table>
<tr><td></td><td>matches the character "n". '\n' matches a newline character. The sequence '\\' matches "\" and "\(" matches "(".</td></tr>
<tr><td>^</td><td>Matches the position at the beginning of the input string. If the **RegExp** object's **Multiline** property is set, ^ also matches the position following '\n' or '\r'.</td></tr>
<tr><td>$</td><td>Matches the position at the end of the input string. If the **RegExp** object's **Multiline** property is set, $ also matches the position preceding '\n' or '\r'.</td></tr>
<tr><td>*</td><td>Matches the preceding subexpression zero or more times. For example, zo* matches "z" and "zoo". * is equivalent to {0,}.</td></tr>
<tr><td>+</td><td>Matches the preceding subexpression one or more times. For example, 'zo+' matches "zo" and "zoo", but not "z". + is equivalent to {1,}.</td></tr>
<tr><td>?</td><td>Matches the preceding subexpression zero or one time. For example, "do(es)?" matches the "do" in "do" or "does". ? is equivalent to {0,1}</td></tr>
<tr><td>{*n*}</td><td>*n* is a nonnegative integer. Matches exactly *n* times. For example, 'o{2}' does not match the 'o' in "Bob," but matches the two o's in "food".</td></tr>
<tr><td>{*n*,}</td><td>*n* is a nonnegative integer. Matches at least *n* times. For example, 'o{2,}' does not match the "o" in "Bob" and matches all the o's in "foooood". 'o{1,}' is equivalent to 'o+'. 'o{0,}' is equivalent to 'o*'.</td></tr>
<tr><td>{*n*,*m*}</td><td>*m* and *n* are nonnegative integers, where *n* <= *m*. Matches at least *n* and at most *m* times. For example, "o{1,3}" matches the first three o's in "fooooood". 'o{0,1}' is equivalent to 'o?'. Note that you cannot put a space between the comma and the numbers.</td></tr>
<tr><td>?</td><td>When this character immediately follows any of the other quantifiers (*, +, ?, {*n*}, {*n*,}, {*n*,*m*}), the matching pattern is non-greedy. A non-greedy pattern matches as little of the searched string as possible, whereas the default greedy pattern matches as much of the searched string as possible. For example, in the string "oooo", 'o+?' matches a single "o", while 'o+' matches all 'o's.</td></tr>
<tr><td>.</td><td>Matches any single character except "\n". To match any character including the '\n', use a pattern such as '[.\n]'.</td></tr>
<tr><td>(*pattern*)</td><td>Matches *pattern* and captures the match. The captured match can be retrieved from the resulting Matches collection, using the **SubMatches** collection in VBScript or the **$0**…**$9** properties in JScript. To match parentheses characters ( ), use '\(' or '\)'.</td></tr>
<tr><td>(?:*pattern*)</td><td>Matches *pattern* but does not capture the match, that is, it is a non-capturing match that is not stored for possible later use. This is useful for combining parts of a pattern with the "or" character (|). For example, 'industr(?:y|ies) is a more economical expression than 'industry|industries'.</td></tr>
<tr><td>(?=*pattern*)</td><td>Positive lookahead matches the search string at any point where a string matching *pattern* begins. This is a non-capturing match, that is, the match is not captured for possible later use. For example 'Windows (?=95|98|NT|2000)' matches "Windows" in "Windows 2000" but not "Windows" in "Windows 3.1". Lookaheads do not consume characters, that is, after a match occurs, the search for the next match begins immediately following the last match, not after the characters that comprised the lookahead.</td></tr>
<tr><td>(?!*pattern*)</td><td>Negative lookahead matches the search string at any point where a string not matching *pattern* begins. This is a non-capturing match, that is, the match is not captured for possible later use. For example 'Windows (?!95|98|NT|2000)'</td></tr>
</table>

	matches "Windows" in "Windows 3.1" but does not match "Windows" in "Windows 2000". Lookaheads do not consume characters, that is, after a match occurs, the search for the next match begins immediately following the last match, not after the characters that comprised the lookahead.
x|*y*	Matches either *x* or *y*. For example, 'z|food' matches "z" or "food". '(z|f)ood' matches "zood" or "food".
[*xyz*]	A character set. Matches any one of the enclosed characters. For example, '[abc]' matches the 'a' in "plain".
[^*xyz*]	A negative character set. Matches any character not enclosed. For example, '[^abc]' matches the 'p' in "plain".
[*a*-*z*]	A range of characters. Matches any character in the specified range. For example, '[a-z]' matches any lowercase alphabetic character in the range 'a' through 'z'.
[^*a*-*z*]	A negative range characters. Matches any character not in the specified range. For example, '[^a-z]' matches any character not in the range 'a' through 'z'.
\b	Matches a word boundary, that is, the position between a word and a space. For example, 'er\b' matches the 'er' in "never" but not the 'er' in "verb".
\B	Matches a nonword boundary. 'er\B' matches the 'er' in "verb" but not the 'er' in "never".
\c*x*	Matches the control character indicated by *x*. For example, \cM matches a Control-M or carriage return character. The value of *x* must be in the range of A-Z or a-z. If not, c is assumed to be a literal 'c' character.
\d	Matches a digit character. Equivalent to [0-9].
\D	Matches a nondigit character. Equivalent to [^0-9].
\f	Matches a form-feed character. Equivalent to \x0c and \cL.
\n	Matches a newline character. Equivalent to \x0a and \cJ.
\r	Matches a carriage return character. Equivalent to \x0d and \cM.
\s	Matches any whitespace character including space, tab, form-feed, etc. Equivalent to [\f\n\r\t\v].
\S	Matches any non-white space character. Equivalent to [^ \f\n\r\t\v].
\t	Matches a tab character. Equivalent to \x09 and \cI.
\v	Matches a vertical tab character. Equivalent to \x0b and \cK.
\w	Matches any word character including underscore. Equivalent to '[A-Za-z0-9_]'.
\W	Matches any nonword character. Equivalent to '[^A-Za-z0-9_]'.
\x*n*	Matches *n*, where *n* is a hexadecimal escape value. Hexadecimal escape values must be exactly two digits long. For example, '\x41' matches "A". '\x041' is equivalent to '\x04' & "1". Allows ASCII codes to be used in regular expressions.
num	Matches *num*, where *num* is a positive integer. A reference back to captured matches. For example, '(.)\1' matches two consecutive identical characters.
n	Identifies either an octal escape value or a backreference. If *n* is preceded by at least *n* captured subexpressions, *n* is a backreference. Otherwise, *n* is an octal escape value if *n* is an octal digit (0-7).
nm	Identifies either an octal escape value or a backreference. If *nm* is preceded by at least *nm* captured subexpressions, *nm* is a backreference. If *nm* is preceded by at least *n* captures, *n* is a backreference followed by literal *m*. If neither of

	the preceding conditions exists, *nm* matches octal escape value *nm* when *n* and *m* are octal digits (0-7).
nml	Matches octal escape value *nml* when *n* is an octal digit (0-3) and *m* and *l* are octal digits (0-7).
\u*n*	Matches *n*, where *n* is a Unicode character expressed as four hexadecimal digits. For example, \u00A9 matches the copyright symbol (©).

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Build a Regular Expression

Regular expressions are constructed in the same way that arithmetic expressions are created. That is, small expressions are combined using a variety of metacharacters and operators to create larger expressions.

You construct a regular expression by putting the various components of the expression pattern between a pair of delimiters. For JScript, the delimiters are a pair of forward slash (/) characters. For example:

```
/expression/
```

For VBScript, a pair of quotation marks ("") delimit regular expressions. For example:

```
"expression"
```

In both of the examples shown above, the regular expression pattern (*expression*) is stored in the **Pattern** property of the **RegExp** object.

The components of a regular expression can be individual characters, sets of characters, ranges of characters, choices between characters, or any combination of all of these components.

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Order of Precedence

Once you have constructed a regular expression, it is evaluated much like an arithmetic expression, that is, it is evaluated from left to right and follows an order of precedence.

The following table illustrates, from highest to lowest, the order of precedence of the various regular expression operators:

Operator(s)	Description
\	Escape
(), (?:), (?=), []	Parentheses and Brackets
*, +, ?, {n}, {n,}, {n,m}	Quantifiers
^, $, *anymetacharacter*	Anchors and Sequences
|	Alternation

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Ordinary Characters

Ordinary characters consist of all those printable and non-printable characters that are not explicitly designated as metacharacters. This includes all upper- and lowercase alphabetic characters, all digits, all punctuation marks, and some symbols.

The simplest form of a regular expression is a single, ordinary character that matches itself in a searched string. For example, the single-character pattern 'A' matches the letter 'A' wherever it appears in the searched string. Here are some examples of single-character regular expression patterns:

```
/a/
/7/
/M/
```

The equivalent VBScript single-character regular expressions are:

```
"a"
"7"
"M"
```

You can combine a number of single characters together to form a larger expression. For example, the following JScript regular expression is nothing more than an expression created by combining the single-character expressions 'a', '7', and 'M'.

```
/a7M/
```

The equivalent VBScript expression is:

```
"a7M"
```

Notice that there is no concatenation operator. All that is required is that you just put one character after another.

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Special Characters

There are a number of metacharacters that require special treatment when trying to match them. To match these special characters, you must first *escape* those characters, that is, precede them with a backslash character (\). The following table shows those special characters and their meanings:

Special Character	Comment
$	Matches the position at the end of an input string. If the **RegExp** object's **Multiline** property is set, $ also matches the position preceding '\n' or '\r'. To match the $ character itself, use \$.
( )	Marks the beginning and end of a subexpression. Subexpressions can be captured for later use. To match these characters, use \(and \).
*	Matches the preceding subexpression zero or more times. To match the * character, use *.
+	Matches the preceding subexpression one or more times. To match the + character, use \+.
.	Matches any single character except the newline character \n. To match ., use \.
[	Marks the beginning of a bracket expression. To match [, use \[.
?	Matches the preceding subexpression zero or one time, or indicates a non-greedy quantifier. To match the ? character, use \?.
\	Marks the next character as either a special character, a literal, a backreference, or an octal escape. For example, 'n' matches the character 'n'. '\n' matches a newline character. The sequence '\\' matches "\" and '\(' matches "(".
^	Matches the position at the beginning of an input string except when used in a bracket expression where it negates the character set. To match the ^ character itself, use \^.
{	Marks the beginning of a quantifier expression. To match {, use \{.
|	Indicates a choice between two items. To match |, use \\|.

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Non-Printable Characters

There are a number of useful non-printing characters that must be used occasionally. The following table shows the escape sequences used to represent those non-printing characters:

Character	Meaning
\c*x*	Matches the control character indicated by *x*. For example, \cM matches a Control-M or carriage return character. The value of *x* must be in the range of A-Z or a-z. If not, c is assumed to be a literal 'c' character.

\f	Matches a form-feed character. Equivalent to \x0c and \cL.
\n	Matches a newline character. Equivalent to \x0a and \cJ.
\r	Matches a carriage return character. Equivalent to \x0d and \cM.
\s	Matches any whitespace character including space, tab, form-feed, etc. Equivalent to [\f\n\r\t\v].
\S	Matches any non-whitespace character. Equivalent to [^ \f\n\r\t\v].
\t	Matches a tab character. Equivalent to \x09 and \cI.
\v	Matches a vertical tab character. Equivalent to \x0b and \cK.

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Character Matching

The period (.) matches any single printing or non-printing character in a string, except a newline character (\n). The following JScript regular expression matches 'aac', 'abc', 'acc', 'adc', and so on, as well as 'a1c', 'a2c', a-c', and a#c':

```
/a.c/
```

The equivalent VBScript regular expression is:

```
"a.c"
```

If you are trying to match a string containing a file name where a period (.) is part of the input string, you do so by preceding the period in the regular expression with a backslash (\) character. To illustrate, the following JScript regular expression matches 'filename.ext':

```
/filename\.ext/
```

For VBScript, the equivalent expression appears as follows:

```
"filename\.ext"
```

These expressions are still pretty limited. They only let you match *any* single character. Many times, it's useful to match specified characters from a list. For example, if you have an input text that contains chapter headings that are expressed numerically as Chapter 1, Chapter 2, etc, you might want to find those chapter headings.

Bracket Expressions

You can create a list of matching characters by placing one or more individual characters within square brackets ([and]). When characters are enclosed in brackets, the list is called a *bracket expression*. Within brackets, as anywhere else, ordinary characters represent themselves, that is, they match an occurrence of themselves in the input text. Most special characters lose their meaning when they occur inside a bracket expression. Here are some exceptions:

- The ']' character ends a list if it's not the first item. To match the ']' character in a list, place it first, immediately following the opening '['.
- The '\' character continues to be the escape character. To match the '\' character, use '\\'.

Characters enclosed in a bracket expression match only a single character for the position in the regular expression where the bracket expression appears. The following JScript regular expression matches 'Chapter 1', 'Chapter 2', 'Chapter 3', 'Chapter 4', and 'Chapter 5':

```
/Chapter [12345]/
```

To match those same chapter heading in VBScript, use the following:

```
"Chapter [12345]"
```

Notice that the word 'Chapter' and the space that follows are fixed in position relative to the characters within brackets. The bracket expression then, is used to specify only the set of characters that matches the single character position immediately following the word 'Chapter' and a space. That is the ninth character position.

If you want to express the matching characters using a range instead of the characters themselves, you can separate the beginning and ending characters in the range using the hyphen (-) character. The character value of the individual characters determines their relative order within a range. The following JScript regular expression contains a range expression that is equivalent to the bracketed list shown above.

```
/Chapter [1-5]/
```

The same expression for VBScript appears as follows:

```
"Chapter [1-5]"
```

When a range is specified in this manner, both the starting and ending values are included in the range. It is important to note that the starting value must precede the ending value in Unicode sort order.

If you want to include the hyphen character in your bracket expression, you must do one of the following:

- Escape it with a backslash:

  ```
  [\-]
  ```

- Put the hyphen character at the beginning or the end of the bracketed list. The following expressions matches all lowercase letters and the hyphen:

  ```
  [-a-z]
  [a-z-]
  ```

- Create a range where the beginning character value is lower than the hyphen character and the ending character value is equal to or greater than the hyphen. Both of the following regular expressions satisfy this requirement:

  ```
  [!--]
  [!-~]
  ```

You can also find all the characters not in the list or range by placing the caret (^) character at the beginning of the list. If the caret character appears in any other position within the list, it matches itself, that is, it has no special meaning. The following JScript regular expression matches chapter headings with numbers greater than 5':

```
/Chapter [^12345]/
```

For VBScript use:

```
"Chapter [^12345]"
```

In the examples shown above, the expression matches any digit character in the ninth position except 1, 2, 3, 4, or 5. So, for example, 'Chapter 7' is a match and so is 'Chapter 9'.

The same expressions above can be represented using the hyphen character (-). For JScript:

```
/Chapter [^1-5]/
```

or for VBScript:

```
"Chapter [^1-5]"
```

A typical use of a bracket expression is to specify matches of any upper- or lowercase alphabetic characters or any digits. The following JScript expression specifies such a match:

```
/[A-Za-z0-9]/
```

The equivalent expression for VBScript is:

```
"[A-Za-z0-9]"
```

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Quantifiers

Sometimes, you don't know how many characters there are to match. In order to accommodate that kind of uncertainty, regular expressions support the concept of quantifiers. These quantifiers let you specify how many times a given component of your regular expression must occur for your match to be true.

The following table illustrates the various quantifiers and their meanings:

Character	Description
*	Matches the preceding subexpression zero or more times. For example, 'zo*' matches "z" and "zoo". * is equivalent to {0,}.
+	Matches the preceding subexpression one or more times. For example, 'zo+' matches "zo" and "zoo", but not "z". + is equivalent to {1,}.

?	Matches the preceding subexpression zero or one time. For example, 'do(es)?' matches the "do" in "do" or "does". ? is equivalent to {0,1}
{*n*}	*n* is a nonnegative integer. Matches exactly *n* times. For example, 'o{2}' does not match the 'o' in "Bob," but matches the two o's in "food".
{*n*,}	*n* is a nonnegative integer. Matches at least *n* times. For example, 'o{2,}' does not match the 'o' in "Bob" and matches all the o's in "foooood". 'o{1,}' is equivalent to 'o+'. 'o{0,}' is equivalent to 'o*'.
{*n*,*m*}	*m* and *n* are nonnegative integers, where *n* <= *m*. Matches at least *n* and at most *m* times. For example, 'o{1,3}' matches the first three o's in "fooooood". 'o{0,1}' is equivalent to 'o?'. Note that you cannot put a space between the comma and the numbers.

With a large input document, chapter numbers could easily exceed nine, so you need a way to handle two or three digit chapter numbers. Quantifiers give you that capability. The following JScript regular expression matches chapter headings with any number of digits:

```
/Chapter [1-9][0-9]*/
```

The following VBScript regular expression performs the identical match:

```
"Chapter [1-9][0-9]*"
```

Notice that the quantifier appears after the range expression. Therefore, it applies to the entire range expression that, in this case, specifies only digits from 0 through 9, inclusive.

The '+' quantifier is not used here because there does not necessarily need to be a digit in the second or subsequent position. The '?' character also is not used because it limits the chapter numbers to only two digits. You want to match at least one digit following 'Chapter' and a space character.

If you know that your chapter numbers are limited to only 99 chapters, you can use the following JScript expression to specify at least one, but not more than 2 digits.

```
/Chapter [0-9]{1,2}/
```

For VBScript, use the following regular expression:

```
"Chapter [0-9]{1,2}"
```

The disadvantage to the expression shown above is that if there is a chapter number greater than 99, it will still only match the first two digits. Another disadvantage is that somebody could create a Chapter 0 and it would match. Better JScript expressions for matching only two digits are the following:

```
/Chapter [1-9][0-9]?/
```

or

```
/Chapter [1-9][0-9]{0,1}/
```

For VBScript, the following expressions are equivalent:

```
"Chapter [1-9][0-9]?"
```

or

```
"Chapter [1-9][0-9]{0,1}"
```

The `'*'`, `'+'`, and `'?'` quantifiers are all what are referred to as *greedy*, that is, they match as much text as possible. Sometimes that's not at all what you want to happen. Sometimes, you just want a minimal match.

Say, for example, you are searching an HTML document for an occurrence of a chapter title enclosed in an H1 tag. That text appears in your document as:

```
<H1>Chapter 1 – Introduction to Regular Expressions</H1>
```

The following expression matches everything from the opening less than symbol (<) to the greater than symbol at the end of the closing H1 tag.

```
/<.*>/
```

The VBScript regular expression is:

```
"<.*>"
```

If all you really wanted to match was the opening H1 tag, the following, non-greedy expression matches only <H1>.

```
/<.*?>/
```

or

```
"<.*?>"
```

By placing the '?' after a '*', '+', or '?' quantifier, the expression is transformed from a greedy to a non-greedy, or minimal, match.

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Anchors

So far, the examples you've seen have been concerned only with finding chapter headings wherever they occur. Any occurrence of the string 'Chapter' followed by a space, followed by a number, could be an actual chapter heading, or it could also be a cross-reference to another chapter. Since true chapter headings always appear at the beginning of a line, you'll need to devise a way to find only the headings and not find the cross-references.

Anchors provide that capability. Anchors allow you to fix a regular expression to either the beginning or end of a line. They also allow you to create regular expressions that occur either within a word or at the beginning or end of a word. The following table contains the list of regular expression anchors and their meanings:

Character	Description
^	Matches the position at the beginning of the input string. If the **RegExp** object's **Multiline** property is set, ^ also matches the position following '\n' or '\r'.
$	Matches the position at the end of the input string. If the **RegExp** object's **Multiline** property is set, $ also matches the position preceding '\n' or '\r'.
\b	Matches a word boundary, that is, the position between a word and a space.
\B	Matches a nonword boundary.

You cannot use a quantifier with an anchor. Since you cannot have more than one position immediately before or after a newline or word boundary, expressions such as '^*' are not permitted.

To match text at the beginning of a line of text, use the '^' character at the beginning of the regular expression. Don't confuse this use of the '^' with the use within a bracket expression. They're definitely not the same.

To match text at the end of a line of text, use the '$' character at the end of the regular expression.

To use anchors when searching for chapter headings, the following JScript regular expression matches a chapter heading with up to two following digits that occurs at the beginning of a line:

```
/^Chapter [1-9][0-9]{0,1}/
```

For VBScript the same regular expressions appears as:

```
"^Chapter [1-9][0-9]{0,1}"
```

Not only does a true chapter heading occur at the beginning of a line, it's also the only thing on the line, so it also must be at the end of a line as well. The following expression ensures that the match you've specified only matches chapters and not cross-references. It does so by creating a regular expression that matches only at the beginning and end of a line of text.

```
/^Chapter [1-9][0-9]{0,1}$/
```

For VBScript use:

```
"^Chapter [1-9][0-9]{0,1}$"
```

Matching word boundaries is a little different but adds a very important capability to regular expressions. A word boundary is the position between a word and a space. A non-word boundary is any other position. The following JScript expression matches the first three characters of the word 'Chapter' because they appear following a word boundary:

```
/\bCha/
```

or for VBScript:

```
"\bCha"
```

The position of the '\b' operator is critical here. If it's positioned at the beginning of a string to be matched, it looks for the match at the beginning of the word; if it's positioned at the end of the string, it looks for the match at the end of the word. For example, the following expressions match 'ter' in the word 'Chapter' because it appears before a word boundary:

```
/ter\b/
```

and

```
"ter\b"
```

The following expressions match 'apt' as it occurs in 'Chapter', but not as it occurs in 'aptitude':

```
/\Bapt/
```

and

```
"\Bapt"
```

That's because 'apt' occurs on a non-word boundary in the word 'Chapter' but on a word boundary in the word 'aptitude'. For the non-word boundary operator, position isn't important because the match isn't relative to the beginning or end of a word.

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Alternation and Grouping

Alternation allows use of the '|' character to allow a choice between two or more alternatives. Expanding the chapter heading regular expression, you can expand it to cover more than just chapter headings. However, it's not as straightforward as you might think. When alternation is used, the largest possible expression on either side of the '|' character is matched. You might think that the following expressions for JScript and VBScript match either 'Chapter' or 'Section' followed by one or two digits occurring at the beginning and ending of a line:

```
/^Chapter|Section [1-9][0-9]{0,1}$/
"^Chapter|Section [1-9][0-9]{0,1}$"
```

Unfortunately, what happens is that the regular expressions shown above match either the word 'Chapter' at the beginning of a line, or 'Section' and whatever numbers follow that, at the end of the line. If the input string is 'Chapter 22', the expression shown above only matches the word 'Chapter'. If the input string is 'Section 22', the expression matches 'Section 22'. But that's not the intent here so there must be a way to make that regular expression more responsive to what you're trying to do and there is.

You can use parentheses to limit the scope of the alternation, that is, make sure that it applies only to the two words, 'Chapter' and 'Section'. However, parentheses are tricky as well, because they are also used to create subexpressions, something that's covered later in the section on subexpressions. By taking the regular expressions shown above and adding parentheses in the appropriate places, you can make the regular expression match either 'Chapter 1' or 'Section 3'.

The following regular expressions use parentheses to group 'Chapter' and 'Section' so the expression works properly. For JScript:

```
/^(Chapter|Section) [1-9][0-9]{0,1}$/
```

For VBScript:

```
"^(Chapter|Section) [1-9][0-9]{0,1}$"
```

These expressions work properly except that an interesting by-product occurs. Placing parentheses around 'Chapter|Section' establishes the proper grouping, but it also causes either of the two matching words to be captured for future use. Since there's only one set of parentheses in the expression shown above, there is only one captured *submatch*. This submatch can be referred to using the **Submatches** collection in VBScript or the **$1-$9** properties of the **RegExp** object in JScript.

Sometimes capturing a submatch is desirable, sometimes it's not. In the examples shown above, all you really want to do is use the parentheses for grouping a choice between the words 'Chapter' or 'Section'. You don't necessarily want to refer to that match later. In fact, unless you really need to capture submatches, don't use them. Your regular expressions will be more efficient since they won't have to take the time and memory to store those submatches.

You can use '?:' before the regular expression pattern inside the parentheses to prevent the match from being saved for possible later use. The following modification of the regular expressions shown above provides the same capability without saving the submatch. For JScript:

```
/^(?:Chapter|Section) [1-9][0-9]{0,1}$/
```

For VBScript:

```
"^(?:Chapter|Section) [1-9][0-9]{0,1}$"
```

In addition to the '?:' metacharacters, there are two other non-capturing metacharacters used for something called *lookahead* matches. A positive lookahead, specified using ?=, matches the search string at any point where a matching regular expression pattern in parentheses begins. A negative lookahead, specified using '?!', matches the search string at any point where a string not matching the regular expression pattern begins.

For example, suppose you have a document containing references to Windows 3.1, Windows 95, Windows 98, and Windows NT. Suppose further that you need to update the document by finding all the references to Windows 95, Windows 98, and Windows NT and changing those reference to Windows 2000. You can use the following JScript regular expression, which is an example of a positive lookahead, to match Windows 95, Windows 98, and Windows NT:

```
/Windows(?=95 |98 |NT )/
```

To make the same match in VBScript, use the following:

```
"Windows(?=95 |98 |NT )"
```

Once the match is found, the search for the next match begins immediately following the matched text, not including the characters included in the look-ahead. For example, if the expressions shown above matched 'Windows 98', the search resumes after 'Windows' not after '98'.

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Backreferences

One of the most important features of regular expressions is the ability to store a part of a matched pattern for later reuse. As you'll recall, placing parentheses around a regular expression pattern or part of a pattern causes that part of the expression to be stored into a temporary buffer. You can override the saving of that part of the regular expression using the non-capturing metacharacters '?:', '?=', or '?!'.

Each captured submatch is stored as it is encountered from left to right in a regular expressions pattern. The buffer numbers where the submatches are stored begin at 1 and continue up to a maximum of 99 subexpressions. Each different buffer can be accessed using '*n*' where *n* is one or two decimal digits identifying a specific buffer.

One of the simplest, most useful applications of back references provides the ability to locate the occurrence of two identical words together in a text. Take the following sentence:

```
Is is the cost of of gasoline going up up?
```

As written, the sentence shown above clearly has a problem with several duplicated words. It would be nice to devise a way to fix that sentence without having to look for duplicates of every single word. The following JScript regular expression uses a single subexpression to do that.

```
/\b([a-z]+) \1\b/gi
```

The equivalent VBScript expression is:

```
"\b([a-z]+) \1\b"
```

The subexpression, in this case, is everything between parentheses. That captured expression includes one or more alphabetic characters, as specified by '[a-z]+'. The second part of the regular expression is the reference to the previously captured submatch, that is, the second occurrence of the word just matched by the parenthetical expression. '\1' is used to specified the first submatch. The word boundary Meta characters ensure that only separate words are detected. If they weren't, a phrase such as "is issued" or "this is" would be incorrectly identified by this expression.

In the JScript expression the global flag ('g') following the regular expression indicates that the expression is applied to as many matches as it can find in the input string. The case insensitivity ('i') flag at the end of the expression specifies the case insensitivity. The multiline flag specifies that potential matches may occur on either side of a newline character. For VBScript, the various flags cannot be set in the expression but must be explicitly set using properties of the **RegExp** object.

Using the regular expression shown above, the following JScript code can use the submatch information to replace an occurrence of two consecutive identical words in a string of text with a single occurrence of the same word:

```
var ss = "Is is the cost of of gasoline going up up?.\n";
var re = /\b([a-z]+) \1\b/gim;       //Create regular expression pattern.
var rv = ss.replace(re,"$1");   //Replace two occurrences with one.
```

The closest equivalent VBScript code appears as follows:

```
Dim ss, re, rv
ss = "Is is the cost of of gasoline going up up?." & vbNewLine
Set re = New RegExp
re.Pattern = "\b([a-z]+) \1\b"
re.Global = True
re.IgnoreCase = True
re.MultiLine = True
rv = re.Replace(ss,"$1")
```

In the VBScript code, notice that the global, case-insensitivity, and multiline flags are set using the appropriately named properties of the **RegExp** object.

The use of the **$1** within the **replace** method refers to the first saved submatch. If you had more than one submatch, you'd refer to them consecutively by **$2**, **$3**, and so on.

Another way that backreferences can be used is to break down a Universal Resource Indicator (URI) into its component parts. Assume that you want to break down the following URI down to the protocol (ftp, http, etc), the domain address, and the page/path:

```
http://msdn.microsoft.com:80/scripting/default.htm
```

The following regular expressions provide that functionality. For JScript:

```
/(\w+):\/\/([^/:]+)(:\d*)?([^# ]*)/
```

For VBScript:

```
"(\w+):\/\/([^/:]+)(:\d*)?([^# ]*)"
```

The first parenthetical subexpression is designed to capture the protocol part of the web address. That subexpression matches any word that precedes a colon and two forward slashes. The second parenthetical subexpression captures the domain address part of the address. That subexpression matches any sequence of characters that does not include '^', '/', or ':' characters. The third parenthetical subexpression captures a website port number, if one is specified. That subexpression matches zero or more digits following a colon. And finally, the fourth parenthetical subexpression captures the path and/or page information specified by the web address. That subexpression matches one or more characters other than '#' or the space character.

Applying the regular expression to the URI shown above, the submatches contain the following:

- **RegExp.$1** contains "http"

- **RegExp.$2** contains "msdn.microsoft.com"
- **RegExp.$3** contains ":80"
- **RegExp.$4** contains "/scripting/default.htm"

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

VBScript Language Reference

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Version Information

The following table lists the version of Microsoft Visual Basic Scripting Edition implemented by host applications.

Host Application	1.0	2.0	3.0	4.0	5.0	5.5	5.6
Microsoft Internet Explorer 3.0	x						
Microsoft Internet Information Server 3.0		x					
Microsoft Internet Explorer 4.0			x				
Microsoft Internet Information Server 4.0			x				
Microsoft Windows Scripting Host 1.0			x				
Microsoft Outlook 98			x				
Microsoft Visual Studio 6.0				x			
Microsoft Internet Explorer 5.0					x		
Microsoft Internet Information Services 5.0					x		
Microsoft Internet Explorer 5.5						x	
Microsoft Visual Studio .NET							x

The following table lists VBScript language features and the version when first introduced.

Language Element	1.0	2.0	3.0	4.0	5.0	5.5	5.6
Abs Function	x						
Addition Operator (+)	x						
And Operator	x						
Array Function		x					
Asc Function	x						

Assignment Operator (=)	x				
Atn Function	x				
Call Statement	x				
CBool Function	x				
CByte Function	x				
CCur Function	x				
CDate Function	x				
CDbl Function	x				
Chr Function	x				
CInt Function	x				
Class Object					x
Class Statement					x
Clear Method	x				
CLng Function	x				
Color Constants		x			
Comparison Constants		x			
Concatenation Operator (&)	x				
Const Statement		x			
Cos Function	x				
CreateObject Function		x			
CSng Function	x				
CStr Function	x				
Date and Time Constants		x			
Date Format Constants		x			
Date Function	x				
DateAdd Function		x			
DateDiff Function		x			
DatePart Function		x			
DateSerial Function	x				
DateValue Function	x				
Day Function	x				
Description Property	x				
Dim Statement	x				
Division Operator (/)	x				

Do...Loop Statement	x				
Empty	x				
Eqv Operator	x				
Erase Statement	x				
Err Object	x				
Eval Function					x
Execute Method					x
Execute Statement					x
ExecuteGlobal Statement					x
Exit Statement	x				
Exp Function	x				
Exponentiation Operator (^)	x				
False	x				
Filter Function		x			
FirstIndex Property					x
Fix Function	x				
For...Next Statement	x				
For Each...Next Statement		x			
FormatCurrency Function		x			
FormatDateTime Function		x			
FormatNumber Function		x			
FormatPercent Function		x			
Function Statement	x				
GetLocale Function					x
GetObject Function		x			
GetRef Function					x
Global Property					x
Hex Function	x				
HelpContext Property		x			
HelpFile Property		x			
Hour Function	x				
If...Then...Else Statement	x				
IgnoreCase Property					x
Imp Operator	x				

Initialize Event			x
InputBox Function	x		
InStr Function	x		
InStrRev Function		x	
Int Function	x		
Integer Division Operator (\)	x		
Is Operator	x		
IsArray Function	x		
IsDate Function	x		
IsEmpty Function	x		
IsNull Function	x		
IsNumeric Function	x		
IsObject Function	x		
Join Function		x	
LBound Function	x		
LCase Function	x		
Left Function	x		
Len Function	x		
Length Property			x
LoadPicture Function		x	
Log Function	x		
LTrim Function	x		
Match Object			x
Matches Collection			x
Mid Function	x		
Minute Function	x		
Miscellaneous Constants		x	
Mod Operator	x		
Month Function	x		
MonthName Function		x	
MsgBox Constants		x	
MsgBox Function	x		
Multiplication Operator (*)	x		
Negation Operator (-)	x		

Not Operator	x		
Now Function	x		
Nothing	x		
Null	x		
Number Property	x		
Oct Function	x		
On Error Statement	x		
Option Explicit Statement	x		
Or Operator	x		
Pattern Property			x
Private Statement		x	
PropertyGet Statement			x
PropertyLet Statement			x
PropertySet Statement			x
Public Statement		x	
Raise Method	x		
Randomize Statement	x		
ReDim Statement	x		
RegExp Object			x
Rem Statement	x		
Replace Function		x	
Replace Method			x
RGB Function		x	
Right Function	x		
Rnd Function	x		
Round Function		x	
RTrim Function	x		
ScriptEngine Function		x	
ScriptEngineBuildVersion Function		x	
ScriptEngineMajorVersion Function		x	
ScriptEngineMinorVersion Function		x	
Second Function	x		
Select Case Statement	x		
Set Statement	x		

SetLocale Function			**x**
Sgn Function	**x**		
Sin Function	**x**		
Source Property	**x**		
Space Function	**x**		
Split Function		**x**	
Sqr Function	**x**		
StrComp Function	**x**		
String Constants		**x**	
String Function	**x**		
StrReverse Function		**x**	
Sub Statement	**x**		
Subtraction Operator (-)	**x**		
Tan Function	**x**		
Terminate Event			**x**
Test Method			**x**
Time Function	**x**		
Timer Function			**x**
TimeSerial Function	**x**		
TimeValue Function	**x**		
Trim Function	**x**		
Tristate Constants		**x**	
True	**x**		
TypeName Function		**x**	
UBound Function	**x**		
UCase Function	**x**		
Value Property			**x**
VarType Constants		**x**	
VarType Function	**x**		
VBScript Constants		**x**	
Weekday Function	**x**		
WeekdayName Function		**x**	
While...Wend Statement	**x**		
With Statement			**x**

Xor Operator	**x**
Year Function	**x**

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

VBScript Constants

A number of useful constants you can use in your code are built into VBScript. Constants provide a convenient way to use specific values without actually having to remember the value itself. Using constants also makes your code more maintainable should the value of any constant ever change. Because these constants are already defined in VBScript, you don't need to explicitly declare them in your code. Simply use them in place of the values they represent.

Here are the various categories of constants provided in VBScript and a brief description of each:

- Color Constants Defines eight basic colors that can be used in scripting.
- Date and Time Constants Defines date and time constants used by various date and time functions.
- Date Format Constants Defines constants used to format dates and times.
- Miscellaneous Constants Defines constants that don't conveniently fit into any other category.
- MsgBox Constants Defines constants used in the **MsgBox** function to describe button visibility, labeling, behavior, and return values.
- String Constants Defines a variety of non-printable characters used in string manipulation.
- Tristate Constants Defines constants used with functions that format numbers.
- VarType Constants Defines the various Variant subtypes.

Requirements

Version 2

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Color Constants

Since these constants are built into VBScript, you don't have to define them before using them. Use them anywhere in your code to represent the values shown for each.

Constant	Value	Description
vbBlack	&h00	Black
vbRed	&hFF	Red
vbGreen	&hFF00	Green
vbYellow	&hFFFF	Yellow
vbBlue	&hFF0000	Blue
vbMagenta	&hFF00FF	Magenta
vbCyan	&hFFFF00	Cyan
vbWhite	&hFFFFFF	White

Requirements

Version 2

See Also

Comparison Constants | Date and Time Constants | Date Format Constants | Miscellaneous Constants | MsgBox Constants | String Constants | Tristate Constants | VarType Constants

© 2001 Microsoft Corporation. All rights reserved.

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Comparison Constants

Since these constants are built into VBScript, you don't have to define them before using them. Use them anywhere in your code to represent the values shown for each.

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

Requirements

Version 2

See Also

Color Constants | Date and Time Constants | Date Format Constants | Miscellaneous Constants | MsgBox Constants | String Constants | Tristate Constants | VarType Constants

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Date and Time Constants

Since these constants are built into VBScript, you don't have to define them before using them. Use them anywhere in your code to represent the values shown for each.

Constant	Value	Description
vbSunday	1	Sunday
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday
vbUseSystemDayOfWeek	0	Use the day of the week specified in your system settings for the first day of the week.
vbFirstJan1	1	Use the week in which January 1 occurs (default).
vbFirstFourDays	2	Use the first week that has at least four days in the new year.
vbFirstFullWeek	3	Use the first full week of the year.

Requirements

Version 2

See Also

Color Constants | Comparison Constants | Date Format Constants | Miscellaneous Constants | MsgBox Constants | String Constants | Tristate Constants | VarType Constants

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Date Format Constants

Since these constants are built into VBScript, you don't have to define them before using them. Use them anywhere in your code to represent the values shown for each.

Constant	Value	Description
vbGeneralDate	0	Display a date and/or time. For real numbers, display a date and time. If there is no fractional part, display only a date. If there is no integer part, display time only. Date and time display is determined by your system settings.
vbLongDate	1	Display a date using the long date format specified in your computer's regional settings.
vbShortDate	2	Display a date using the short date format specified in your computer's regional settings.
vbLongTime	3	Display a time using the long time format specified in your computer's regional settings.
vbShortTime	4	Display a time using the short time format specified in your computer's regional settings.

Requirements

Version 2

See Also

Color Constants | Comparison Constants | Date and Time Constants | Miscellaneous Constants | MsgBox Constants | String Constants | Tristate Constants | VarType Constants

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Miscellaneous Constants

Since this constant is built into VBScript, you don't have to define it before using it. Use it anywhere in your code to represent the values shown.

Constant	Value	Description
vbObjectError	-2147221504	User-defined error numbers should be greater than this value, for example, `Err.Raise Number = vbObjectError + 1000`

Requirements

Version 2

See Also

Color Constants | Comparison Constants | Date and Time Constants | Date Format Constants | MsgBox Constants | String Constants | Tristate Constants | VarType Constants

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

MsgBox Constants

The following constants are used with the **MsgBox** function to identify what buttons and icons appear on a message box and which button is the default. In addition, the modality of the **MsgBox** can be specified. Since these constants are built into VBScript, you don't have to define them before using them. Use them anywhere in your code to represent the values shown for each.

Constant	Value	Description
vbOKOnly	0	Display **OK** button only.
vbOKCancel	1	Display **OK** and **Cancel** buttons.

vbAbortRetryIgnore	2	Display **Abort**, **Retry**, and **Ignore** buttons.
vbYesNoCancel	3	Display **Yes**, **No**, and **Cancel** buttons.
vbYesNo	4	Display **Yes** and **No** buttons.
vbRetryCancel	5	Display **Retry** and **Cancel** buttons.
vbCritical	16	Display Critical Message icon.
vbQuestion	32	Display **Warning Query** icon.
vbExclamation	48	Display Warning Message icon.
vbInformation	64	Display Information Message icon.
vbDefaultButton1	0	First button is the default.
vbDefaultButton2	256	Second button is the default.
vbDefaultButton3	512	Third button is the default.
vbDefaultButton4	768	Fourth button is the default.
vbApplicationModal	0	Application modal. The user must respond to the message box before continuing work in the current application.
vbSystemModal	4096	System modal. On Win16 systems, all applications are suspended until the user responds to the message box. On Win32 systems, this constant provides an application modal message box that always remains on top of any other programs you may have running.

The following constants are used with the **MsgBox** function to identify which button a user has selected. These constants are only available when your project has an explicit reference to the appropriate type library containing these constant definitions. For VBScript, you must explicitly declare these constants in your code.

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

Requirements

Version 2

See Also

Color Constants | Comparison Constants | Date and Time Constants | Date Format Constants | Miscellaneous Constants | String Constants | Tristate Constants | VarType Constants

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

String Constants

Since these constants are built into VBScript, you don't have to define them before using them. Use them anywhere in your code to represent the values shown for each.

Constant	Value	Description
vbCr	Chr(13)	Carriage return.
VbCrLf	Chr(13) & Chr(10)	Carriage return–linefeed combination.
vbFormFeed	Chr(12)	Form feed; not useful in Microsoft Windows.
vbLf	Chr(10)	Line feed.
vbNewLine	Chr(13) & Chr(10) or Chr(10)	Platform-specific newline character; whatever is appropriate for the platform.
vbNullChar	Chr(0)	Character having the value 0.
vbNullString	String having value 0	Not the same as a zero-length string (""); used for calling external procedures.
vbTab	Chr(9)	Horizontal tab.
vbVerticalTab	Chr(11)	Vertical tab; not useful in Microsoft Windows.

Requirements

Version 2

See Also

Color Constants | Comparison Constants | Date and Time Constants | Date Format Constants | Miscellaneous Constants | MsgBox Constants | Tristate Constants | VarType Constants

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Tristate Constants

Since these constants are built into VBScript, you don't have to define them before using them. Use them anywhere in your code to represent the values shown for each.

Constant	Value	Description
vbUseDefault	-2	Use default from computer's regional settings.
vbTrue	-1	True
vbFalse	0	False

Requirements

Version 2

See Also

Color Constants | Comparison Constants | Date and Time Constants | Date Format Constants | Miscellaneous Constants | MsgBox Constants | String Constants | VarType Constants

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

VarType Constants

These constants are only available when your project has an explicit reference to the appropriate type library containing these constant definitions. For VBScript, you must explicitly declare these constants in your code.

Constant	Value	Description
vbEmpty	0	Uninitialized (default)
vbNull	1	Contains no valid data
vbInteger	2	Integer subtype
vbLong	3	Long subtype
vbSingle	4	Single subtype
vbSingle	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 (used only for arrays of variants)
vbDataObject	13	Data access object
vbDecimal	14	Decimal subtype
vbByte	17	Byte subtype
vbArray	8192	Array

Requirements

Version 2

See Also

Color Constants | Comparison Constants | Date and Time Constants | Date Format Constants | Miscellaneous Constants | MsgBox Constants | String Constants | Tristate Constants

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Errors

In This Section

VBScript Run-time Errors

VBScript Syntax Errors

Related Sections

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

VBScript Run-time Errors

VBScript run-time errors are errors that result when your VBScript script attempts to perform an action that the system cannot execute. VBScript run-time errors occur while your script is being executed; when variable expressions are being evaluated, and memory is being dynamic allocated.

Error Number	Description
429	ActiveX component can't create object
507	An exception occurred
449	Argument not optional
17	Can't perform requested operation
430	Class doesn't support Automation
506	Class not defined
11	Division by zero
48	Error in loading DLL
5020	Expected ')' in regular expression
5019	Expected ']' in regular expression
432	File name or class name not found during Automation operation
92	For loop not initialized
5008	Illegal assignment
51	Internal error
505	Invalid or unqualified reference
481	Invalid picture
5	Invalid procedure call or argument
5021	Invalid range in character set
94	Invalid use of Null
448	Named argument not found
447	Object doesn't support current locale setting
445	Object doesn't support this action
438	Object doesn't support this property or method
451	Object not a collection
504	Object not safe for creating
503	Object not safe for initializing
502	Object not safe for scripting
424	Object required
91	Object variable not set

7	Out of Memory
28	Out of stack space
14	Out of string space
6	Overflow
35	Sub or function not defined
9	Subscript out of range
5017	Syntax error in regular expression
462	The remote server machine does not exist or is unavailable
10	This array is fixed or temporarily locked
13	Type mismatch
5018	Unexpected quantifier
500	Variable is undefined
458	Variable uses an Automation type not supported in VBScript
450	Wrong number of arguments or invalid property assignment

See Also

VBScript Syntax Errors

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

VBScript Syntax Errors

VBScript syntax errors are errors that result when the structure of one of your VBScript statements violates one or more of the grammatical rules of the VBScript scripting language. VBScript syntax errors occur during the program compilation stage, before the program has begun to be executed.

Error Number	Description
1052	Cannot have multiple default property/method in a Class
1044	Cannot use parentheses when calling a Sub
1053	Class initialize or terminate do not have arguments
1058	'Default' specification can only be on Property Get
1057	'Default' specification must also specify 'Public'
1005	Expected '('
1006	Expected ')'
1011	Expected '='
1021	Expected 'Case'
1047	Expected 'Class'
1025	Expected end of statement
1014	Expected 'End'
1023	Expected expression
1015	Expected 'Function'
1010	Expected identifier
1012	Expected 'If
1046	Expected 'In'
1026	Expected integer constant
1049	Expected Let or Set or Get in property declaration
1045	Expected literal constant
1019	Expected 'Loop'
1020	Expected 'Next'
1050	Expected 'Property'
1022	Expected 'Select'
1024	Expected statement
1016	Expected 'Sub'
1017	Expected 'Then'
1013	Expected 'To'
1018	Expected 'Wend'
1027	Expected 'While' or 'Until'
1028	Expected 'While,' 'Until,' or end of statement
1029	Expected 'With'
1030	Identifier too long

1014	Invalid character
1039	Invalid 'exit' statement
1040	Invalid 'for' loop control variable
1013	Invalid number
1037	Invalid use of 'Me' keyword
1038	'loop' without 'do'
1048	Must be defined inside a Class
1042	Must be first statement on the line
1041	Name redefined
1051	Number of arguments must be consistent across properties specification
1001	Out of Memory
1054	Property Set or Let must have at least one argument
1002	Syntax error
1055	Unexpected 'Next'
1015	Unterminated string constant

See Also

VBScript Run-time Errors

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Events

In This Section

Initialize Event

Terminate Event

Related Sections

VBScript Langauge Reference

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Initialize Event

Occurs when an instance of the associated class is created.

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

The *statements* part consists of zero or more code statements to be run when the class is initialized.

Remarks

The following example illustrates the use of the **Initialize** event.

```
Class TestClass
   Private Sub Class_Initialize   ' Setup Initialize event.
      MsgBox("TestClass started")
   End Sub
   Private Sub Class_Terminate   ' Setup Terminate event.
      MsgBox("TestClass terminated")
   End Sub
End Class
```

```
Set X = New TestClass   ' Create an instance of TestClass.
Set X = Nothing   ' Destroy the instance.
```

Requirements

Version 5

See Also

Class Object | Class Statement | Terminate Event

Applies To: Class Object

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Terminate Event

Occurs when an instance of the associated class is terminated.

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

The *statements* part consists of zero or more code statements to be run when the class is initialized.

Remarks

The following example illustrates the use of the **Terminate** event.

```
Class TestClass
   Private Sub Class_Initialize   ' Setup Initialize event.
      MsgBox("TestClass started")
   End Sub
    Private Sub Class_Terminate   ' Setup Terminate event.
      MsgBox("TestClass terminated")
   End Sub
End Class
Set X = New TestClass   ' Create an instance of TestClass.
Set X = Nothing   ' Destroy the instance.
```

Requirements

Version 5

See Also

Class Object | Class Statement | Initialize Event

Applies To: Class Object

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Functions

The following table contains the VBScript functions.

Abs	Array	Asc	Atn
CBool	CByte	CCur	CDate
CDbl	Chr	CInt	CLng

Conversions	Cos	CreateObject	CSng
Date	DateAdd	DateDiff	DatePart
DateSerial	DateValue	Day	Derived Maths
Eval	Exp	Filter	FormatCurrency
FormatDateTime	FormatNumber	FormatPercent	GetLocale
GetObject	GetRef	Hex	Hour
InputBox	InStr	InStrRev	Int, Fixs
IsArray	IsDate	IsEmpty	IsNull
IsNumeric	IsObject	Join	LBound
LCase	Left	Len	LoadPicture
Log	LTrim; RTrim; and Trims	Maths	Mid
Minute	Month	MonthName	MsgBox
Now	Oct	Replace	RGB
Right	Rnd	Round	ScriptEngine
ScriptEngineBuildVersion	ScriptEngineMajorVersion	ScriptEngineMinorVersion	Second
SetLocale	Sgn	Sin	Space
Split	Sqr	StrComp	String
Tan	Time	Timer	TimeSerial
TimeValue	TypeName	UBound	UCase
VarType	Weekday	WeekdayName	Year

Related Sections

VBScript Langauge Reference

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Abs Function

Returns the absolute value of a number.

```
Abs(number)
```

The *number* argument can be any valid numeric expression. If *number* contains Null, **Null** is returned; if it is an uninitialized variable, zero is returned.

Remarks

The absolute value of a number is its unsigned magnitude. For example, **Abs(**-1**)** and **Abs(**1**)** both return 1.

The following example uses the **Abs** function to compute the absolute value of a number:

```
Dim MyNumber
MyNumber = Abs(50.3)  ' Returns 50.3.
MyNumber = Abs(-50.3) ' Returns 50.3.
```

Requirements

Version 1

See Also

Sgn Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Array Function

Returns a **Variant** containing an array.

Array(*arglist***)**

The required *arglist* argument is a comma-delimited list of values that are assigned to the elements of an array contained with the **Variant**. If no arguments are specified, an array of zero length is created.

Remarks

The notation used to refer to an element of an array consists of the variable name followed by parentheses containing an index number indicating the desired element. In the following example, the first statement creates a variable named A. The second statement assigns an array to variable A. The last statement assigns the value contained in the second array element to another variable.

```
Dim A
A = Array(10,20,30)
B = A(2)   ' B is now 30.
```

> **Note** A variable that is not declared as an array can still contain an array. Although a **Variant** variable containing an array is conceptually different from an array variable containing **Variant** elements, the array elements are accessed in the same way.

Requirements

Version 2

See Also

Dim Statement

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Asc Function

Returns the ANSI character code corresponding to the first letter in a string.

```
Asc(string)
```

The *string* argument is any valid string expression. If the *string* contains no characters, a run-time error occurs.

Remarks

In the following example, **Asc** returns the ANSI character code of the first letter of each string:

```
Dim MyNumber
MyNumber = Asc("A")       ' Returns 65.
MyNumber = Asc("a")       ' Returns 97.
MyNumber = Asc("Apple")   ' Returns 65.
```

> **Note** The **AscB** function is used with byte data contained in a string. Instead of returning the character code for the first character, **AscB** returns the first byte. **AscW** is provided for 32-bit platforms that use Unicode characters. It returns the Unicode (wide) character code, thereby avoiding the conversion from Unicode to ANSI.

Requirements

Version 1

See Also

Chr Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Atn Function

Returns the arctangent of a number.

`Atn(``number`**`)`**

The *number* argument can be any valid numeric expression.

Remarks

The **Atn** function takes the ratio of two sides of a right triangle (*number*) and returns the corresponding angle in radians. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle. The range of the result is -pi /2 to pi/2 radians.

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

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

```
Dim pi
pi = 4 * Atn(1)   ' Calculate the value of pi.
```

> **Note** **Atn** is the inverse trigonometric function of **Tan**, which takes an angle as its argument and returns the ratio of two sides of a right triangle. Do not confuse **Atn** with the cotangent, which is the simple inverse of a tangent (1/tangent).

Requirements

Version 1

See Also

Cos Function | Derived Math Functions | Sin Function | Tan Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

CBool Function

Returns an expression that has been converted to a **Variant** of subtype **Boolean**.

CBool(expression**)**

The *expression* argument is any valid expression.

Remarks

If *expression* is zero, **False** is returned; otherwise, **True** is returned. If *expression* can't be interpreted as a numeric value, a run-time error occurs.

The following example uses the **CBool** function to convert an expression to a **Boolean**. If the expression evaluates to a nonzero value, **CBool** returns **True**; otherwise, it returns **False**.

```
Dim A, B, Check
A = 5: B = 5            ' Initialize variables.
Check = CBool(A = B)    ' Check contains True.
A = 0                   ' Define variable.
Check = CBool(A)        ' Check contains False.
```

Requirements

Version 1

See Also

CByte Function | CCur Function | CDate Function | CDbl Function | CInt Function | CLng Function | CSng Function | CStr Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

CByte Function

Returns an expression that has been converted to a **Variant** of subtype **Byte**.

`CByte(``expression`**`)`**

The *expression* argument is any valid expression.

Remarks

In general, you can document your code using the subtype conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use **CByte** to force byte arithmetic in cases where currency, single-precision, double-precision, or integer arithmetic normally would occur.

Use the **CByte** function to provide internationally aware conversions from any other data type to a **Byte** subtype. For example, different decimal separators are properly recognized depending on the locale setting of your system, as are different thousand separators.

If *expression* lies outside the acceptable range for the byte subtype, an error occurs. The following example uses the **CByte** function to convert an expression to a byte:

```
Dim MyDouble, MyByte
MyDouble = 125.5678        ' MyDouble is a Double.
MyByte = CByte(MyDouble)   ' MyByte contains 126.
```

Requirements

Version 1

See Also

CBool Function | CCur Function | CDate Function | CDbl Function | CInt Function | CLng Function | CSng Function | CStr Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

CCur Function

Returns an expression that has been converted to a **Variant** of subtype **Currency**.

`CCur(``expression`**`)`**

The *expression* argument is any valid expression.

Remarks

In general, you can document your code using the subtype conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, 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.

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

```
Dim MyDouble, MyCurr
MyDouble = 543.214588         ' MyDouble is a Double.
```

```
MyCurr = CCur(MyDouble * 2)   ' Convert result of MyDouble * 2 (1086.429176) to a Currency (1086.4292).
```

Requirements

Version 1

See Also

CBool Function | CByte Function | CDate Function | CDbl Function | CInt Function | CLng Function | CSng Function | CStr Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

CDate Function

Returns an expression that has been converted to a **Variant** of subtype **Date**.

CDate(*date*)

The *date* argument is any valid date expression.

Remarks

Use the **IsDate** function to determine if *date* can be converted to a date or time. **CDate** recognizes date literals and time literals as well as some numbers that fall within the range of acceptable dates. When converting a number to a date, the whole number portion is converted to a date. Any fractional part of the number is converted to a time of day, starting at midnight.

CDate recognizes date formats according to the locale setting of your system. The correct order of day, month, and year may not be determined if it is provided in a format other than one of the recognized date settings. In addition, a long date format is not recognized if it also contains the day-of-the-week string.

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.

```
MyDate = "October 19, 1962"   ' Define date.
MyShortDate = CDate(MyDate)   ' Convert to Date data type.
MyTime = "4:35:47 PM"         ' Define time.
MyShortTime = CDate(MyTime)   ' Convert to Date data type.
```

Requirements

Version 1

See Also

IsDate Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

CDbl Function

Returns an expression that has been converted to a **Variant** of subtype **Double**.

```
CDbl(expression)
```

The *expression* argument is any valid expression.

Remarks

In general, you can document your code using the subtype conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use **CDbl** or **CSng** to force double-precision or single-precision arithmetic in cases where currency or integer arithmetic normally would occur.

Use the **CDbl** function to provide internationally aware conversions from any other data type to a **Double** subtype. For example, different decimal separators and thousands separators are properly recognized depending on the locale setting of your system.

This example uses the **CDbl** function to convert an expression to a **Double**.

```
Dim MyCurr, MyDouble
MyCurr = CCur(234.456784)              ' MyCurr is a Currency (234.4567).
MyDouble = CDbl(MyCurr * 8.2 * 0.01)   ' Convert result to a Double (19.2254576).
```

Requirements

Version 1

See Also

CBool Function | CByte Function | CCur Function | CDate Function | CInt Function | CLng Function | CSng Function | CStr Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Chr Function

Returns the character associated with the specified ANSI character code.

```
Chr(charcode)
```

The *charcode* argument is a number that identifies a character.

Remarks

Numbers from 0 to 31 are the same as standard, nonprintable ASCII codes. For example, **Chr(**10**)** returns a linefeed character.

The following example uses the **Chr** function to return the character associated with the specified character code:

```
Dim MyChar
MyChar = Chr(65)   ' Returns A.
MyChar = Chr(97)   ' Returns a.
MyChar = Chr(62)   ' Returns >.
MyChar = Chr(37)   ' Returns %.
```

> **Note** The **ChrB** function is used with byte data contained in a string. Instead of returning a character, which may be one or two bytes, **ChrB** always returns a single byte. **ChrW** is provided for 32-bit platforms that use Unicode characters. Its argument is a Unicode (wide) character code, thereby avoiding the conversion from ANSI to Unicode.

Requirements

Version 1

See Also

Asc Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

CInt Function

Returns an expression that has been converted to a **Variant** of subtype **Integer**.

```
CInt(expression)
```

The *expression* argument is any valid expression.

Remarks

In general, you can document your code using the subtype conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use **CInt** or **CLng** to force integer arithmetic in cases where currency, single-precision, or double-precision arithmetic normally would occur.

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.

If *expression* lies outside the acceptable range for the Integer subtype, an error occurs.

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

```
Dim MyDouble, MyInt
MyDouble = 2345.5678     ' MyDouble is a Double.
MyInt = CInt(MyDouble)   ' MyInt contains 2346.
```

> **Note** **CInt** differs from the **Fix** and **Int** functions, which truncate, rather than round, the fractional part of a number. When the fractional part is exactly 0.5, the **CInt** function always rounds it to the nearest even number. For example, 0.5 rounds to 0, and 1.5 rounds to 2.

Requirements

Version 1

See Also

CBool Function | CByte Function | CCur Function | CDate Function | CDbl Function | CLng Function | CSng Function | CStr Function | Int, Fix Functions

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

CLng Function

Returns an expression that has been converted to a **Variant** of subtype **Long**.

```
CLng(expression)
```

The *expression* argument is any valid expression.

Remarks

In general, you can document your code using the subtype conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, 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.

If *expression* lies outside the acceptable range for the Long subtype, an error occurs.

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

```
Dim MyVal1, MyVal2, MyLong1, MyLong2
MyVal1 = 25427.45: MyVal2 = 25427.55   ' MyVal1, MyVal2 are Doubles.
MyLong1 = CLng(MyVal1)   ' MyLong1 contains 25427.
MyLong2 = CLng(MyVal2)   ' MyLong2 contains 25428.
```

> **Note** **CLng** differs from the **Fix** and **Int** functions, which truncate, rather than round, the fractional part of a number. When the fractional part is exactly 0.5, the **CLng** function always rounds it to the nearest even number. For example, 0.5 rounds to 0, and 1.5 rounds to 2.

Requirements

Version 1

See Also

CBool Function | CByte Function | CCur Function | CDate Function | CDbl Function | CInt Function | CSng Function | CStr Function | Int, Fix Functions

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Conversion Functions

Asc Function

CBool Function

CByte Function

CCur Function

CDate Function

CDbl Function

Chr Function

CInt Function

[CLng Function](#)

[CSng Function](#)

[CStr Function](#)

[Hex Function](#)

[Oct Function](#)

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Cos Function

Returns the cosine of an angle.

`Cos(``number`**`)`**

The *number* argument can be any valid numeric expression that expresses an angle in radians.

Remarks

The **Cos** function takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side adjacent to the angle divided by the length of the hypotenuse. The result lies in the range -1 to 1.

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

The following example uses the **Cos** function to return the cosine of an angle:

```
Dim MyAngle, MySecant
MyAngle = 1.3                  ' Define angle in radians.
MySecant = 1 / Cos(MyAngle)    ' Calculate secant.
```

Requirements

Version 1

See Also

Atn Function | Derived Math Functions | Sin Function | Tan Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

CreateObject Function

Creates and returns a reference to an Automation object.

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

Arguments

servername
    Required. The name of the application providing the object.
typename
    Required. The type or class of the object to create.
location
    Optional. The name of the network server where the object is to be created.

Remarks

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 ExcelSheet
Set ExcelSheet = CreateObject("Excel.Sheet")
```

This code starts the application that creates the object (in this case, a Microsoft Excel spreadsheet). Once an object is created, refer to it in code using the object variable you defined. As shown in the following example, you can access properties and methods of the new object using the object variable, `ExcelSheet`, and other Excel objects, including the Application object and the ActiveSheet.Cells collection:

```
' Make Excel visible through the Application object.
ExcelSheet.Application.Visible = True
' Place some text in the first cell of the sheet.
ExcelSheet.ActiveSheet.Cells(1,1).Value = "This is column A, row 1"
' Save the sheet.
ExcelSheet.SaveAs "C:\DOCS\TEST.XLS"
' Close Excel with the Quit method on the Application object.
ExcelSheet.Application.Quit
' Release the object variable.
Set ExcelSheet = Nothing
```

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.

The following code returns the version number of an instance of Excel running on a remote network computer named "myserver":

```
Function GetVersion
   Dim XLApp
   Set XLApp = CreateObject("Excel.Application", "MyServer")
   GetVersion = XLApp.Version
End Function
```

An error occurs if the specified remote server does not exist or cannot be found.

Requirements

Version 2

See Also

GetObject Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

CSng Function

Returns an expression that has been converted to a **Variant** of subtype **Single**.

`CSng(``expression`**`)`**

The *expression* argument is any valid expression.

Remarks

In general, you can document your code using the data type conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, 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.

If *expression* lies outside the acceptable range for the Single subtype, an error occurs.

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

```
Dim MyDouble1, MyDouble2, MySingle1, MySingle2   ' MyDouble1, MyDouble2 are Doubles.
MyDouble1 = 75.3421115: MyDouble2 = 75.3421555
MySingle1 = CSng(MyDouble1)   ' MySingle1 contains 75.34211.
MySingle2 = CSng(MyDouble2)   ' MySingle2 contains 75.34216.
```

Requirements

Version 1

See Also

CBool Function | CByte Function | CCur Function | CDate Function | CDbl Function | CInt Function | CLng Function | CStr Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

CStr Function

Returns an expression that has been converted to a **Variant** of subtype **String**.

```
CStr(expression)
```

The *expression* argument is any valid expression.

Remarks

In general, you can document your code using the data type conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, 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.

The data in *expression* determines what is returned according to the following table:

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	A **String** containing the number.

The following example uses the **CStr** function to convert a numeric value to a **String**:

```
Dim MyDouble, MyString
MyDouble = 437.324         ' MyDouble is a Double.
MyString = CStr(MyDouble)   ' MyString contains "437.324".
```

Requirements

Version 1

See Also

CBool Function | CByte Function | CCur Function | CDate Function | CDbl Function | CInt Function | CLng Function | CSng Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Date Function

Returns the current system date.

```
Date
```

Remarks

The following example uses the **Date** function to return the current system date:

```
Dim MyDate
MyDate = Date   ' MyDate contains the current system date.
```

Requirements

Version 1

See Also

CDate Function | Now Function | Time Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

DateAdd Function

Returns a date to which a specified time interval has been added.

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

Arguments

interval
: Required. String expression that is the interval you want to add. See Settings section for values.

number
: Required. Numeric expression that is the number of interval you want to add. The numeric expression can either be positive, for dates in the future, or negative, for dates in the past.

date
: Required. **Variant** or literal representing the date to which *interval* is added.

Settings

The *interval* argument can have the following values:

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

Remarks

You can use the **DateAdd** function to add or subtract a specified time interval from a date. For example, you can use **DateAdd** to calculate a date 30 days from today or a time 45 minutes from now. To add days to *date*, you can use Day of Year ("y"), Day ("d"), or Weekday ("w").

The **DateAdd** function won't return an invalid date. The following example adds one month to January 31:

```
NewDate = DateAdd("m", 1, "31-Jan-95")
```

In this case, **DateAdd** returns 28-Feb-95, not 31-Feb-95. If *date* is 31-Jan-96, it returns 29-Feb-96 because 1996 is a leap year.

If the calculated date would precede the year 100, an error occurs.

If number isn't a **Long** value, it is rounded to the nearest whole number before being evaluated.

Requirements

Version 2

See Also

DateDiff Function | DatePart Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

DateDiff Function

Returns the number of intervals between two dates.

DateDiff(interval, date1, date2 [,firstdayofweek[, firstweekofyear]]**)**

The **DateDiff** function syntax has these parts:

Arguments

interval
: Required. String expression that is the interval you want to use to calculate the differences between *date1* and *date2*. See Settings section for values.

date1, *date2*

Required. Date expressions. Two dates you want to use in the calculation.

firstdayofweek

Optional. Constant that specifies the day of the week. If not specified, Sunday is assumed. See Settings section for values.

firstweekofyear

Optional. Constant that specifies the first week of the year. If not specified, the first week is assumed to be the week in which January 1 occurs. See Settings section for values.

Settings

The *interval* argument can have the following values:

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

The *firstdayofweek* argument can have the following values:

Constant	Value	Description
vbUseSystemDayOfWeek	0	Use National Language Support (NLS) API setting.
vbSunday	1	Sunday (default)
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

The *firstweekofyear* argument can have the following values:

Constant	Value	Description
vbUseSystem	0	Use National Language Support (NLS) API setting.
vbFirstJan1	1	Start with the week in which January 1 occurs (default).
vbFirstFourDays	2	Start with the week that has at least four days in the new year.
vbFirstFullWeek	3	Start with the first full week of the new year.

Remarks

You can use the **DateDiff** function to determine how many specified time intervals exist between two dates. For example, you might use **DateDiff** to calculate the number of days between two dates, or the number of weeks between today and the end of the year.

To calculate the number of days between *date1* and *date2*, you can use either Day of year ("y") or Day ("d"). When *interval* is Weekday ("w"), **DateDiff** returns the number of weeks between the two dates. If *date1* falls on a Monday, **DateDiff** counts the number of Mondays until *date2*. It counts *date2* but not *date1*. If *interval* is Week ("ww"), however, the **DateDiff** function returns the number of calendar weeks between the two dates. It counts the number of Sundays between *date1* and *date2*. **DateDiff** counts *date2* if it falls on a Sunday; but it doesn't count *date1*, even if it does fall on a Sunday.

If *date1* refers to a later point in time than *date2*, the **DateDiff** function returns a negative number.

The *firstdayofweek* argument affects calculations that use the "w" and "ww" interval symbols.

If *date1* or *date2* is a date literal, the specified year becomes a permanent part of that date. However, if *date1* or *date2* is enclosed in quotation marks (" ") and you omit the year, the current year is inserted in your code each time the *date1* or *date2* expression is evaluated. This makes it possible to write code that can be used in different years.

When comparing December 31 to January 1 of the immediately succeeding year, **DateDiff** for Year ("yyyy") returns 1 even though only a day has elapsed.

The following example uses the **DateDiff** function to display the number of days between a given date and today:

```
Function DiffADate(theDate)
   DiffADate = "Days from today: " & DateDiff("d", Now, theDate)
End Function
```

Requirements

Version 2

See Also

DateAdd Function | DatePart Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

DatePart Function

Returns the specified part of a given date.

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

Arguments

interval
: Required. String expression that is the interval of time you want to return. See Settings section for values.

date
: Required. Date expression you want to evaluate.

firstdayof week
: Optional. Constant that specifies the day of the week. If not specified, Sunday is assumed. See Settings section for values.

firstweekofyear
: Optional. Constant that specifies the first week of the year. If not specified, the first week is assumed to be the week in which January 1 occurs. See Settings section for values.

Settings

The *interval* argument can have the following values:

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

The *firstdayofweek* argument can have the following values:

Constant	Value	Description
vbUseSystemDayOfWeek	0	Use National Language Support (NLS) API setting.
vbSunday	1	Sunday (default)
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

The *firstweekofyear* argument can have the following values:

Constant	Value	Description
vbUseSystem	0	Use National Language Support (NLS) API setting.
vbFirstJan1	1	Start with the week in which January 1 occurs (default).
vbFirstFourDays	2	Start with the week that has at least four days in the new year.
vbFirstFullWeek	3	Start with the first full week of the new year.

Remarks

You can use the **DatePart** function to evaluate a date and return a specific interval of time. For example, you might use **DatePart** to calculate the day of the week or the current hour.

The *firstdayofweek* argument affects calculations that use the "w" and "ww" interval symbols.

If *date* is a date literal, the specified year becomes a permanent part of that date. However, if *date* is enclosed in quotation marks (" "), and you omit the year, the current year is inserted in your code each time the *date* expression is evaluated. This makes it possible to write code that can be used in different years.

This example takes a date and, using the **DatePart** function, displays the quarter of the year in which it occurs.

```
Function GetQuarter(TheDate)
   GetQuarter = DatePart("q", TheDate)
End Function
```

Requirements

Version 2

See Also

DateAdd Function | DateDiff Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

DateSerial Function

Returns a **Variant** of subtype **Date** for a specified year, month, and day.

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

Arguments

year
Number between 100 and 9999, inclusive, or a numeric expression.

month
Any numeric expression.

day
Any numeric expression.

Remarks

To specify a date, such as December 31, 1991, the range of numbers for each **DateSerial** argument should be in the accepted range for the unit; that is, 1–31 for days and 1–12 for months. However, you can also specify relative dates for each argument using any numeric expression that represents some number of days, months, or years before or after a certain date.

The following example uses numeric expressions instead of absolute date numbers. Here the **DateSerial** function returns a date that is the day before the first day (1 – 1) of two months before August (8 – 2) of 10 years before 1990 (1990 – 10); in other words, May 31, 1980.

```
Dim MyDate1, MyDate2
MyDate1 = DateSerial(1970, 1, 1)   ' Returns January 1, 1970.
MyDate2 = DateSerial(1990 - 10, 8 - 2, 1 - 1)   ' Returns May 31, 1980.
```

For the *year* argument, values between 0 and 99, inclusive, are interpreted as the years 1900–1999. For all other *year* arguments, use a complete four-digit year (for example, 1800).

When any argument exceeds the accepted range for that argument, it increments to the next larger unit as appropriate. For example, if you specify 35 days, it is evaluated as one month and some number of days, depending on where in the year it is applied. However, if any single argument is outside the range -32,768 to 32,767, or if the date specified by the three arguments, either directly or by expression, falls outside the acceptable range of dates, an error occurs.

Requirements

[Version 1]

See Also

Date Function | DateValue Function | Day Function | Month Function | Now Function | TimeSerial Function | TimeValue Function | Weekday Function | Year Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

DateValue Function

Returns a **Variant** of subtype **Date**.

```
DateValue(date)
```

The *date* argument is normally a string expression representing a date from January 1, 100 through December 31, 9999. However, *date* can also be any expression that can represent a date, a time, or both a date and time, in that range.

Remarks

If the *date* argument includes time information, **DateValue** doesn't return it. However, if *date* includes invalid time information (such as "89:98"), an error occurs.

If *date* is a string that includes only numbers separated by valid date separators, **DateValue** recognizes the order for month, day, and year according to the short date format you specified for your system. **DateValue** also recognizes unambiguous dates that contain month names, either in long or abbreviated form. For example, in addition to recognizing 12/30/1991 and 12/30/91, **DateValue** also recognizes December 30, 1991 and Dec 30, 1991.

If the year part of *date* is omitted, **DateValue** uses the current year from your computer's system date.

The following example uses the **DateValue** function to convert a string to a date. You can also use date literals to directly assign a date to a

Variant variable, for example, MyDate = #9/11/63#.

```
Dim MyDate
MyDate = DateValue("September 11, 1963")   ' Return a date.
```

Requirements

Version 1

See Also

CDate Function | DateSerial Function | Day Function | Month Function | Now Function | TimeSerial Function | TimeValue Function | Weekday Function | Year Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Day Function

Returns a whole number between 1 and 31, inclusive, representing the day of the month.

```
Day(date)
```

The *date* argument is any expression that can represent a date. If *date* contains Null, **Null** is returned.

The following example uses the **Day** function to obtain the day of the month from a specified date:

```
Dim MyDay
MyDay = Day("October 19, 1962")   ' MyDay contains 19.
```

Requirements

Version 1

See Also

Date Function | Hour Function | Minute Function | Month Function | Now Function | Second Function | Weekday Function | Year Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Derived Math Functions

The following non-intrinsic math functions can be derived from the intrinsic math functions:

Function	Derived equivalents
Secant	Sec(X) = 1 / Cos(X)
Cosecant	Cosec(X) = 1 / Sin(X)
Cotangent	Cotan(X) = 1 / Tan(X)
Inverse Sine	Arcsin(X) = Atn(X / Sqr(-X * X + 1))
Inverse Cosine	Arccos(X) = Atn(-X / Sqr(-X * X + 1)) + 2 * Atn(1)
Inverse Secant	Arcsec(X) = Atn(X / Sqr(X * X - 1)) + Sgn((X) -1) * (2 * Atn(1))
Inverse Cosecant	Arccosec(X) = Atn(X / Sqr(X * X - 1)) + (Sgn(X) - 1) * (2 * Atn(1))
Inverse Cotangent	Arccotan(X) = Atn(X) + 2 * Atn(1)
Hyperbolic Sine	HSin(X) = (Exp(X) - Exp(-X)) / 2
Hyperbolic Cosine	HCos(X) = (Exp(X) + Exp(-X)) / 2
Hyperbolic Tangent	HTan(X) = (Exp(X) - Exp(-X)) / (Exp(X) + Exp(-X))
Hyperbolic Secant	HSec(X) = 2 / (Exp(X) + Exp(-X))

Hyperbolic Cosecant	HCosec(X) = 2 / (Exp(X) - Exp(-X))
Hyperbolic Cotangent	HCotan(X) = (Exp(X) + Exp(-X)) / (Exp(X) - Exp(-X))
Inverse Hyperbolic Sine	HArcsin(X) = Log(X + Sqr(X * X + 1))
Inverse Hyperbolic Cosine	HArccos(X) = Log(X + Sqr(X * X - 1))
Inverse Hyperbolic Tangent	HArctan(X) = Log((1 + X) / (1 - X)) / 2
Inverse Hyperbolic Secant	HArcsec(X) = Log((Sqr(-X * X + 1) + 1) / X)
Inverse Hyperbolic Cosecant	HArccosec(X) = Log((Sgn(X) * Sqr(X * X + 1) +1) / X)
Inverse Hyperbolic Cotangent	HArccotan(X) = Log((X + 1) / (X - 1)) / 2
Logarithm to base N	LogN(X) = Log(X) / Log(N)

See Also

Atn Function | Cos Function | Exp Function | Log Function | Sin Function | Sqr Function | Tan Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Eval

Evaluates an expression and returns the result.

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

Arguments

result

Optional. Variable to which return value assignment is made. If *result* is not specified, consider using the **Execute** statement instead.

expression

Required. String containing any legal VBScript expression.

Remarks

In VBScript, *x* = *y* can be interpreted two ways. The first is as an assignment statement, where the value of *y* is assigned to *x*. The second interpretation is as an expression that tests if *x* and *y* have the same value. If they do, *result* is **True**; if they are not, *result* is **False**. The **Eval** method always uses the second interpretation, whereas the **Execute** statement always uses the first.

> **Note** In Microsoft® JScript™, no confusion exists between assignment and comparison, because the assignment operator (=) is different from the comparison operator (==).

The following example illustrates the use of the **Eval** function:

```
Sub GuessANumber
   Dim Guess, RndNum
   RndNum = Int((100) * Rnd(1) + 1)
   Guess = CInt(InputBox("Enter your guess:",,0))
   Do
      If Eval("Guess = RndNum") Then
         MsgBox "Congratulations! You guessed it!"
         Exit Sub
      Else
         Guess = CInt(InputBox("Sorry! Try again.",,0))
      End If
   Loop Until Guess = 0
End Sub
```

Requirements

Version 5

See Also

Execute Statement

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Exp Function

Returns *e* (the base of natural logarithms) raised to a power.

```
Exp(number)
```

The *number* argument can be any valid numeric expression.

Remarks

If the value of *number* exceeds 709.782712893, an error occurs. The constant *e* is approximately 2.718282.

> **Note** The **Exp** function complements the action of the **Log** function and is sometimes referred to as the antilogarithm.

The following example uses the **Exp** function to return e raised to a power:

```
Dim MyAngle, MyHSin    ' Define angle in radians.
MyAngle = 1.3    ' Calculate hyperbolic sine.
MyHSin = (Exp(MyAngle) - Exp(-1 * MyAngle)) / 2
```

Requirements

Version 1

See Also

Derived Math Functions | Log Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Filter Function

Returns a zero-based array containing a subset of a string array based on a specified filter criteria.

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

Arguments

InputStrings
    Required. One-dimensional array of strings to be searched.
Value
    Required. String to search for.
Include
    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.
Compare
    Optional. Numeric value indicating the kind of string comparison to use. See Settings section for values.

Settings

The *Compare* argument can have the following values:

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

Remarks

If no matches of *Value* are found within *InputStrings*, **Filter** returns an empty array. An error occurs if *InputStrings* is **Null** or is not a one-dimensional array.

The array returned by the **Filter** function contains only enough elements to contain the number of matched items.

The following example uses the **Filter** function to return the array containing the search criteria "Mon":

```
Dim MyIndex
Dim MyArray (3)
MyArray(0) = "Sunday"
MyArray(1) = "Monday"
MyArray(2) = "Tuesday"
MyIndex = Filter(MyArray, "Mon") ' MyIndex(0) contains "Monday".
```

Requirements

Version 2

See Also

Replace Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

FormatCurrency Function

Returns an expression formatted as a currency value using the currency symbol defined in the system control panel.

```
FormatCurrency(Expression[,NumDigitsAfterDecimal [,IncludeLeadingDigit [,UseParensForNegativeNumbers [,GroupDigits
```

Arguments

Expression
  Required. Expression to be formatted.

NumDigitsAfterDecimal

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.

IncludeLeadingDigit

Optional. Tristate constant that indicates whether or not a leading zero is displayed for fractional values. See Settings section for values.

UseParensForNegativeNumbers

Optional. Tristate constant that indicates whether or not to place negative values within parentheses. See Settings section for values.

GroupDigits

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.

Settings

The IncludeLeadingDigit, UseParensForNegativeNumbers, and GroupDigits arguments have the following settings:

Constant	Value	Description
TristateTrue	-1	True
TristateFalse	0	False
TristateUseDefault	-2	Use the setting from the computer's regional settings.

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.

> **Note** All settings information comes from the Regional Settings Currency tab, except leading zero, which comes from the Number tab.

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

```
Dim MyCurrency
MyCurrency = FormatCurrency(1000)   ' MyCurrency contains $1000.00.
```

Requirements

Version 2

See Also

FormatDateTime Function | FormatNumber Function | FormatPercent Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

FormatDateTime Function

Returns an expression formatted as a date or time.

`FormatDateTime(`*`Date`*`[, `*`NamedFormat`*`]`**`)`**

Arguments

Date
    Required. Date expression to be formatted.
NamedFormat
    Optional. Numeric value that indicates the date/time format used. If omitted, **vbGeneralDate** is used.

Settings

The *NamedFormat* argument has the following settings:

Constant	Value	Description
vbGeneralDate	0	Display a date and/or time. If there is a date part, display it as a short date. If there is a time part, display it as a long time. If present, both parts are displayed.
vbLongDate	1	Display a date using the long date format specified in your computer's regional settings.
vbShortDate	2	Display a date using the short date format specified in your computer's regional settings.

vbLongTime	3	Display a time using the time format specified in your computer's regional settings.
vbShortTime	4	Display a time using the 24-hour format (hh:mm).

Remarks

The following example uses the **FormatDateTime** function to format the expression as a long date and assign it to MyDateTime:

```
Function GetCurrentDate
   ' FormatDateTime formats Date in long date.
   GetCurrentDate = FormatDateTime(Date, 1)
End Function
```

Requirements

Version 2

See Also

FormatCurrency Function | FormatNumber Function | FormatPercent Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

FormatNumber Function

Returns an expression formatted as a number.

```
FormatNumber(Expression [,NumDigitsAfterDecimal [,IncludeLeadingDigit [,UseParensForNegativeNumbers [,GroupDigits]
```

Arguments

Expression
Required. Expression to be formatted.
NumDigitsAfterDecimal
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.
IncludeLeadingDigit
Optional. Tristate constant that indicates whether or not a leading zero is displayed for fractional values. See Settings section for values.
UseParensForNegativeNumbers
Optional. Tristate constant that indicates whether or not to place negative values within parentheses. See Settings section for values.
GroupDigits
Optional. Tristate constant that indicates whether or not numbers are grouped using the group delimiter specified in the control panel. See Settings section for values.

Settings

The IncludeLeadingDigit, UseParensForNegativeNumbers, and GroupDigits arguments have the following settings:

Constant	Value	Description
TristateTrue	-1	True
TristateFalse	0	False
TristateUseDefault	-2	Use the setting from the computer's regional settings.

Remarks

When one or more of the optional arguments are omitted, the values for omitted arguments are provided by the computer's regional settings.

Note All settings information comes from the Regional Settings Number tab.

The following example uses the **FormatNumber** function to format a number to have four decimal places:

```
Function FormatNumberDemo
   Dim MyAngle, MySecant, MyNumber
   MyAngle = 1.3   ' Define angle in radians.
   MySecant = 1 / Cos(MyAngle)   ' Calculate secant.
   FormatNumberDemo = FormatNumber(MySecant,4) ' Format MySecant to four decimal places.
End Function
```

Requirements

Version 2

See Also

FormatCurrency Function | FormatDateTime Function | FormatPercent Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

FormatPercent Function

Returns an expression formatted as a percentage (multiplied by 100) with a trailing % character.

```
FormatPercent(Expression[,NumDigitsAfterDecimal [,IncludeLeadingDigit [,UseParensForNegativeNumbers [,GroupDigits]
```

The **FormatPercent** function syntax has these parts:

Arguments

Expression
: Required. Expression to be formatted.

NumDigitsAfterDecimal
: 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.

IncludeLeadingDigit
: Optional. Tristate constant that indicates whether or not a leading zero is displayed for fractional values. See Settings section for values.

UseParensForNegativeNumbers
: Optional. Tristate constant that indicates whether or not to place negative values within parentheses. See Settings section for values.

GroupDigits

Optional. Tristate constant that indicates whether or not numbers are grouped using the group delimiter specified in the control panel. See Settings section for values.

Settings

The IncludeLeadingDigit, UseParensForNegativeNumbers, and GroupDigits arguments have the following settings:

Constant	Value	Description
TristateTrue	-1	True
TristateFalse	0	False
TristateUseDefault	-2	Use the setting from the computer's regional settings.

Remarks

When one or more optional arguments are omitted, the values for the omitted arguments are provided by the computer's regional settings.

Note All settings information comes from the Regional Settings Number tab.

The following example uses the **FormatPercent** function to format an expression as a percent:

```
Dim MyPercent
MyPercent = FormatPercent(2/32) ' MyPercent contains 6.25%.
```

Requirements

Version 2

See Also

FormatCurrency Function | FormatDateTime Function | FormatNumber Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

GetLocale Function

Returns the current locale ID value.

```
GetLocale()
```

Remarks

A *locale* is a set of user preference information related to the user's language, country/region, and cultural conventions. The *locale* determines such things as keyboard layout, alphabetic sort order, as well as date, time, number, and currency formats.

The return value can be any of the 32-bit values shown in the Locale ID chart:

The following example illustrates the use of the **GetLocale** function. To use this code, paste the entire example between the <BODY> tags of a standard HTML page.

```
Enter Date in UK format: <input type="text" id="UKDate" size="20"><p>
Here's the US equivalent: <input type="text" id="USdate" size="20"><p>
<input type="button" value="Convert" id="button1"><p>
Enter a price in German:   <input type="text" id="GermanNumber" size="20">
<p>
Here's the UK equivalent: <input type="text" id="USNumber" size="20"><p>
<input type="button" value="Convert" id="button2"><p>

<script language="vbscript">
Dim currentLocale
' Get the current locale
currentLocale = GetLocale

Sub Button1_onclick
  Dim original
  original = SetLocale("en-gb")
  mydate = CDate(UKDate.value)
  ' IE always sets the locale to US English so use the
  ' currentLocale variable to set the locale to US English
```

```
  original = SetLocale(currentLocale)
  USDate.value = FormatDateTime(mydate,vbShortDate)
End Sub

Sub button2_onclick
  Dim original
  original = SetLocale("de")
  myvalue = CCur(GermanNumber.value)
  original = SetLocale("en-gb")
  USNumber.value = FormatCurrency(myvalue)
End Sub

</script>
```

See Also

SetLocale Function | Locale ID (LCID) Chart

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

GetObject Function

Returns a reference to an Automation object from a file.

GetObject([*pathname*] [, *class*]**)**

Arguments

pathname
 Optional; String. Full path and name of the file containing the object to retrieve. If *pathname* is omitted, *class* is required.

class

Optional; String. Class of the object.

The *class* argument uses the syntax *appname.objectype* and has these parts:

Arguments

appname
Required; String. Name of the application providing the object.
objectype
Required; String. Type or class of object to create.

Remarks

Use the **GetObject** function to access an Automation object from a file and assign the object to an object variable. Use the **Set** statement to assign the object returned by **GetObject** to the object variable. For example:

```
Dim CADObject
Set CADObject = GetObject("C:\CAD\SCHEMA.CAD")
```

When this code is executed, the application associated with the specified pathname is started and the object in the specified file is activated. If *pathname* is a zero-length string (""), **GetObject** returns a new object instance of the specified type. If the *pathname* argument is omitted, **GetObject** returns a currently active object of the specified type. If no object of the specified type exists, an error occurs.

Some applications allow you to activate part of a file. Add an exclamation point (!) to the end of the file name and follow it with a string that identifies the part of the file you want to activate. For information on how to create this string, see the documentation for the application that created the object.

For example, in a drawing application you might have multiple layers to a drawing stored in a file. You could use the following code to activate a layer within a drawing called `SCHEMA.CAD`:

```
Set LayerObject = GetObject("C:\CAD\SCHEMA.CAD!Layer3")
```

If you don't specify the object's class, Automation determines the application to start and the object to activate, based on the file name you provide. Some files, however, may support more than one class of object. For example, a drawing might support three different types of objects: an Application object, a Drawing object, and a Toolbar object, all of which are part of the same file. To specify which object in a file you want to activate, use the optional *class* argument. For example:

```
Dim MyObject
Set MyObject = GetObject("C:\DRAWINGS\SAMPLE.DRW", "FIGMENT.DRAWING")
```

In the preceding example, `FIGMENT` is the name of a drawing application and `DRAWING` is one of the object types it supports. Once an object is activated, you reference it in code using the object variable you defined. In the preceding example, you access properties and methods of the new object using the object variable `MyObject`. For example:

```
MyObject.Line 9, 90
MyObject.InsertText 9, 100, "Hello, world."
MyObject.SaveAs "C:\DRAWINGS\SAMPLE.DRW"
```

> **Note** Use the **GetObject** function when there is a current instance of the object or if you want to create the object with a file already loaded. If there is no current instance, and you don't want the object started with a file loaded, use the **CreateObject** function.

If an object has registered itself as a single-instance object, only one instance of the object is created, no matter how many times **CreateObject** is executed. With a single-instance object, **GetObject** always returns the same instance when called with the zero-length string ("") syntax, and it causes an error if the *pathname* argument is omitted.

Requirements

Version 5

See Also

CreateObject Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

GetRef Function

Returns a reference to a procedure that can be bound to an event.

```
Set object.eventname = GetRef(procname)
```

Arguments

object
: Required. Name of the object with which *event* is associated.

event
: Required. Name of the event to which the function is to be bound.

procname
: Required. String containing the name of the **Sub** or **Function** procedure being associated with the *event*.

Remarks

The **GetRef** function allows you to connect a VBScript procedure (**Function** or **Sub**) to any available event on your DHTML (Dynamic HTML) pages. The DHTML object model provides information about what events are available for its various objects.

In other scripting and programming languages, the functionality provided by **GetRef** is referred to as a function pointer, that is, it points to the address of a procedure to be executed when the specified event occurs.

The following example illustrates the use of the **GetRef** function.

```
<SCRIPT LANGUAGE="VBScript">

Function GetRefTest()
   Dim Splash
   Splash = "GetRefTest Version 1.0"   & vbCrLf
   Splash = Splash & Chr(169) & " YourCompany 1999 "
   MsgBox Splash
End Function

Set Window.Onload = GetRef("GetRefTest")
</SCRIPT>
```

Requirements

Version 5

See Also

Function Statement | Set Statement | Sub Statement

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Hex Function

Returns a string representing the hexadecimal value of a number.

```
Hex(number)
```

The *number* argument is any valid expression.

Remarks

If *number* is not already a whole number, it is rounded to the nearest whole number before being evaluated.

If *number* is	Hex returns
Null	Null.
Empty	Zero (0).
Any other number	Up to eight hexadecimal characters.

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

16 in hexadecimal notation.

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

```
Dim MyHex
MyHex = Hex(5)   ' Returns 5.
MyHex = Hex(10)   ' Returns A.
MyHex = Hex(459)   ' Returns 1CB.
```

Requirements

Version 1

See Also

Oct Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Hour Function

Returns a whole number between 0 and 23, inclusive, representing the hour of the day.

```
Hour(time)
```

The *time* argument is any expression that can represent a time. If *time* contains Null, **Null** is returned.

The following example uses the **Hour** function to obtain the hour from the current time:

```
Dim MyTime, MyHour
MyTime = Now
MyHour = Hour(MyTime) ' MyHour contains the number representing
                      ' the current hour.
```

Requirements

Version 1

See Also

Day Function | Minute Function | Now Function | Second Function | Time Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

InputBox Function

Displays a prompt in a dialog box, waits for the user to input text or click a button, and returns the contents of the text box.

InputBox(prompt[, title][, default][, xpos][, ypos][, helpfile, context]**)**

Arguments

prompt
String expression displayed as the message in the dialog box. The maximum length of *prompt* is approximately 1024 characters, depending on the width of the characters used. If *prompt* consists of more than one line, you can separate the lines using a carriage return character (**Chr(**13**)**), a linefeed character (**Chr(**10**)**), or carriage return–linefeed character combination (**Chr(**13**) & Chr(**10**)**) between each line.

title

String expression displayed in the title bar of the dialog box. If you omit *title*, the application name is placed in the title bar.

default

String expression displayed in the text box as the default response if no other input is provided. If you omit *default*, the text box is displayed empty.

xpos

Numeric expression that specifies, in twips, the horizontal distance of the left edge of the dialog box from the left edge of the screen. If *xpos* is omitted, the dialog box is horizontally centered.

ypos

Numeric expression that specifies, in twips, the vertical distance of the upper edge of the dialog box from the top of the screen. If *ypos* is omitted, the dialog box is vertically positioned approximately one-third of the way down the screen.

helpfile

String expression that identifies the Help file to use to provide context-sensitive Help for the dialog box. If *helpfile* is provided, *context* must also be provided.

context

Numeric expression that identifies the Help context number assigned by the Help author to the appropriate Help topic. If *context* is provided, *helpfile* must also be provided.

Remarks

When both *helpfile* and *context* are supplied, a Help button is automatically added to the dialog box.

If the user clicks **OK** or presses **ENTER**, the **InputBox** function returns whatever is in the text box. If the user clicks **Cancel**, the function returns a zero-length string ("").

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

```
Dim Input
Input = InputBox("Enter your name")
MsgBox ("You entered: " & Input)
```

Requirements

Version 1

See Also

MsgBox Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

InStr Function

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

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

Arguments

start
: Optional. Numeric expression that sets the starting position for each search. If omitted, search begins at the first character position. If *start* contains Null, an error occurs. The *start* argument is required if *compare* is specified.

string1
: Required. String expression being searched.

string2
: Required. String expression searched for.

compare
: 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.

Settings

The *compare* argument can have the following values:

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

Return Values

The **InStr** function returns the following values:

If	InStr returns
string1 is zero-length	0
string1 is **Null**	Null
string2 is zero-length	start
string2 is **Null**	Null
string2 is not found	0
string2 is found within *string1*	Position at which match is found
start > **Len(**string2**)**	0

Remarks

The following examples use **InStr** to search a string:

```
Dim SearchString, SearchChar, MyPos
SearchString ="XXpXXpXXPXXP"   ' String to search in.
SearchChar = "P"   ' Search for "P".
MyPos = Instr(4, SearchString, SearchChar, 1)   ' A textual comparison starting at position 4. Returns 6.
MyPos = Instr(1, SearchString, SearchChar, 0)   ' A binary comparison starting at position 1. Returns 9.
MyPos = Instr(SearchString, SearchChar)   ' Comparison is binary by default (last argument is omitted). Returns 9.
MyPos = Instr(1, SearchString, "W")   ' A binary comparison starting at position 1. Returns 0 ("W" is not found).
```

> **Note** The **InStrB** function is used with byte data contained in a string. Instead of returning the character position of the first occurrence of one string within another, **InStrB** returns the byte position.

Requirements

Version 1

See Also

InStrRev Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

InStrRev Function

Returns the position of an occurrence of one string within another, from the end of string.

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

Arguments

string1
  Required. String expression being searched.
string2
  Required. String expression being searched for.
start
  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 *start* contains Null, an error occurs.
compare
  Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. If omitted, a binary comparison is performed. See Settings section for values.

Settings

The *compare* argument can have the following values:

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

Return Values

InStrRev returns the following values:

If	InStrRev returns
string1 is zero-length	0
string1 is **Null**	Null
string2 is zero-length	start
string2 is **Null**	Null
string2 is not found	0
string2 is found within *string1*	Position at which match is found
start > **Len(**string2**)**	0

Remarks

The following examples use the **InStrRev** function to search a string:

```
Dim SearchString, SearchChar, MyPos
SearchString ="XXpXXpXXPXXP"   ' String to search in.
SearchChar = "P"   ' Search for "P".
MyPos = InstrRev(SearchString, SearchChar, 10, 0)   ' A binary comparison starting at position 10. Returns 9.
MyPos = InstrRev(SearchString, SearchChar, -1, 1)   ' A textual comparison starting at the last position. Returns
MyPos = InstrRev(SearchString, SearchChar, 8)   ' Comparison is binary by default (last argument is omitted). Retu
```

Note The syntax for the **InStrRev** function is not the same as the syntax for the **InStr** function.

Requirements

Version 2

See Also

InStr Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Int, Fix Functions

Returns the integer portion of a number.

```
Int(number)
Fix(number)
```

The *number* argument can be any valid numeric expression. If *number* contains Null, **Null** is returned.

Remarks

Both **Int** and **Fix** remove the fractional part of *number* and return the resulting integer value.

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.

Fix(*number***)** is equivalent to:

```
Sgn(number) * Int(Abs(number))
```

The following examples illustrate how the **Int** and **Fix** functions return integer portions of numbers:

```
MyNumber = Int(99.8)    ' Returns 99.
MyNumber = Fix(99.2)    ' Returns 99.
MyNumber = Int(-99.8)   ' Returns -100.
MyNumber = Fix(-99.8)   ' Returns -99.
MyNumber = Int(-99.2)   ' Returns -100.
MyNumber = Fix(-99.2)   ' Returns -99.
```

Requirements

Version 1

See Also

CInt Function | Round Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

IsArray Function

Returns a Boolean value indicating whether a variable is an array.

```
IsArray(varname)
```

The *varname* argument can be any variable.

Remarks

IsArray returns **True** if the variable is an array; otherwise, it returns **False**. **IsArray** is especially useful with variants containing arrays.

The following example uses the **IsArray** function to test whether `MyVariable` is an array:

```
Dim MyVariable
Dim MyArray(3)
MyArray(0) = "Sunday"
MyArray(1) = "Monday"
MyArray(2) = "Tuesday"
MyVariable = IsArray(MyArray) ' MyVariable contains "True".
```

Requirements

Version 1

See Also

IsDate Function | IsEmpty Function | IsNull Function | IsNumeric Function | IsObject Function | VarType Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

IsDate Function

Returns a Boolean value indicating whether an expression can be converted to a date.

IsDate(expression**)**

The *expression* argument can be any date expression or string expression recognizable as a date or time.

Remarks

IsDate returns **True** if the expression is a date or can be converted to a valid date; otherwise, it 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.

The following example uses the **IsDate** function to determine whether an expression can be converted to a date:

```
Dim MyDate, YourDate, NoDate, MyCheck
MyDate = "October 19, 1962": YourDate = #10/19/62#: NoDate = "Hello"
MyCheck = IsDate(MyDate)   ' Returns True.
MyCheck = IsDate(YourDate)   ' Returns True.
```

```
MyCheck = IsDate(NoDate)   ' Returns False.
```

Requirements

Version 1

See Also

CDate Function | IsArray Function | IsEmpty Function | IsNull Function | IsNumeric Function | IsObject Function | VarType Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

IsEmpty Function

Returns a Boolean value indicating whether a variable has been initialized.

```
IsEmpty(expression)
```

The *expression* argument can be any expression. However, because **IsEmpty** is used to determine if individual variables are initialized, the *expression* argument is most often a single variable name.

Remarks

IsEmpty returns **True** if the variable is uninitialized, or is explicitly set to Empty; otherwise, it returns **False**. **False** is always returned if *expression* contains more than one variable.

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

```
Dim MyVar, MyCheck
```

```
MyCheck = IsEmpty(MyVar)   ' Returns True.
MyVar = Null   ' Assign Null.
MyCheck = IsEmpty(MyVar)   ' Returns False.
MyVar = Empty   ' Assign Empty.
MyCheck = IsEmpty(MyVar)   ' Returns True.
```

Requirements

Version 1

See Also

IsArray Function | IsDate Function | IsNull Function | IsNumeric Function | IsObject Function | VarType Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

IsNull Function

Returns a Boolean value that indicates whether an expression contains no valid data (Null).

```
IsNull(expression)
```

The *expression* argument can be any expression.

Remarks

IsNull returns **True** if *expression* is **Null**, that is, it contains no valid data; otherwise, **IsNull** returns **False**. If *expression* consists of more than one variable, **Null** in any constituent variable causes **True** to be returned for the entire expression.

The **Null** value indicates that the variable contains no valid data. **Null** is not the same as Empty, which indicates that a variable has not yet been initialized. It is also not the same as a zero-length string (""), which is sometimes referred to as a null string.

> **Caution** Use the **IsNull** function to determine whether an expression contains a **Null** value. Expressions that you might expect to evaluate to **True** under some circumstances, such as If Var = Null and If Var <> Null, are always **False**. This is because any expression containing a **Null** is itself **Null**, and therefore, **False**.

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

```
Dim MyVar, MyCheck
MyCheck = IsNull(MyVar)   ' Returns False.
MyVar = Null   ' Assign Null.
MyCheck = IsNull(MyVar)   ' Returns True.
MyVar = Empty   ' Assign Empty.
MyCheck = IsNull(MyVar)   ' Returns False.
```

Requirements

Version 1

See Also

IsArray Function | IsDate Function | IsEmpty Function | IsNumeric Function | IsObject Function | VarType Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

IsNumeric Function

Returns a Boolean value indicating whether an expression can be evaluated as a number.

```
IsNumeric(expression)
```

The *expression* argument can be any expression.

Remarks

IsNumeric returns **True** if the entire *expression* is recognized as a number; otherwise, it returns **False**. **IsNumeric** returns **False** if *expression* is a date expression.

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

```
Dim MyVar, MyCheck
MyVar = 53   ' Assign a value.
MyCheck = IsNumeric(MyVar)   ' Returns True.
MyVar = "459.95"   ' Assign a value.
MyCheck = IsNumeric(MyVar)   ' Returns True.
MyVar = "45 Help"   ' Assign a value.
MyCheck = IsNumeric(MyVar)   ' Returns False.
```

Requirements

Version 1

See Also

IsArray Function | IsDate Function | IsEmpty Function | IsNull Function | IsObject Function | VarType Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

IsObject Function

Returns a Boolean value indicating whether an expression references a valid Automation object.

IsObject(*expression*)

The *expression* argument can be any expression.

Remarks

IsObject returns **True** if *expression* is a variable of **Object** subtype or a user-defined object; otherwise, it returns **False**.

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

```
Dim MyInt, MyCheck, MyObject
Set MyObject = Me
MyCheck = IsObject(MyObject)   ' Returns True.
MyCheck = IsObject(MyInt)   ' Returns False.
```

Requirements

Version 1

See Also

IsArray Function | IsDate Function | IsEmpty Function | IsNull Function | IsNumeric Function | Set Statement | VarType Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Join Function

Returns a string created by joining a number of substrings contained in an array.

```
Join(list[, delimiter])
```

Arguments

list
Required. One-dimensional array containing substrings to be joined.
delimiter
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.

Remarks

The following example uses the **Join** function to join the substrings of `MyArray`:

```
Dim MyString
Dim MyArray(3)
MyArray(0) = "Mr."
MyArray(1) = "John "
MyArray(2) = "Doe "
MyArray(3) = "III"
MyString = Join(MyArray) ' MyString contains "Mr. John Doe III".
```

Requirements

Version 2

See Also

Split Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

LBound Function

Returns the smallest available subscript for the indicated dimension of an array.

`LBound(``arrayname[, dimension]`**`)`**

Arguments

arrayname
: Name of the array variable; follows standard variable naming conventions.

dimension
: Whole number indicating which dimension's lower bound is returned. Use 1 for the first dimension, 2 for the second, and so on. If *dimension* is omitted, 1 is assumed.

Remarks

The **LBound** function is used with the **UBound** function to determine the size of an array. Use the **UBound** function to find the upper limit of an array dimension.

The lower bound for any dimension is always 0.

Requirements

Version 1

See Also

Dim Statement | ReDim Statement | UBound Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

LCase Function

Returns a string that has been converted to lowercase.

```
LCase(string)
```

The *string* argument is any valid string expression. If *string* contains Null, **Null** is returned.

Remarks

Only uppercase letters are converted to lowercase; all lowercase letters and non-letter characters remain unchanged.

The following example uses the **LCase** function to convert uppercase letters to lowercase:

```
Dim MyString
Dim LCaseString
MyString = "VBScript"
LCaseString = LCase(MyString)   ' LCaseString contains "vbscript".
```

Requirements

Version 1

See Also

UCase Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Left Function

Returns a specified number of characters from the left side of a string.

```
Left(string, length)
```

Arguments

string
: String expression from which the leftmost characters are returned. If *string* 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 *string*, the entire string is returned.

Remarks

To determine the number of characters in *string*, use the **Len** function.

The following example uses the **Left** function to return the first three characters of `MyString`:

```
Dim MyString, LeftString
MyString = "VBSCript"
LeftString = Left(MyString, 3) ' LeftString contains "VBS".
```

> **Note** The **LeftB** function is used with byte data contained in a string. Instead of specifying the number of characters to return, *length* specifies the number of bytes.

Requirements

[Version 1]

See Also

Len Function | Mid Function | Right Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Len Function

Returns the number of characters in a string or the number of bytes required to store a variable.

```
Len(string | varname)
```

Arguments

string
Any valid string expression. If *string* contains Null, **Null** is returned.
varname
Any valid variable name. If *varname* contains **Null**, **Null** is returned.

Remarks

The following example uses the **Len** function to return the number of characters in a string:

```
Dim MyString
MyString = Len("VBSCRIPT") ' MyString contains 8.
```

> **Note** The **LenB** function is used with byte data contained in a string. Instead of returning the number of characters in a string, **LenB** returns the number of bytes used to represent that string.

Requirements

Version 1

See Also

InStr Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

LoadPicture Function

Returns a picture object. Available only on 32-bit platforms.

`LoadPicture(`*`picturename`*`)`

The *picturename* argument is a string expression that indicates the name of the picture file to be loaded.

Remarks

Graphics formats recognized by **LoadPicture** include bitmap (.bmp) files, icon (.ico) files, run-length encoded (.rle) files, metafile (.wmf) files, enhanced metafiles (.emf), GIF (.gif) files, and JPEG (.jpg) files.

Requirements

Version 2

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Log Function

Returns the natural logarithm of a number.

```
Log(number)
```

The *number* argument can be any valid numeric expression greater than 0.

Remarks

The natural logarithm is the logarithm to the base *e*. The constant *e* is approximately 2.718282.

You can calculate base-*n* logarithms for any number *x* by dividing the natural logarithm of *x* by the natural logarithm of *n* as follows:

```
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
```

Requirements

Version 1

See Also

Derived Math Functions | Exp Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

LTrim; RTrim; and Trim Functions

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

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

The *string* argument is any valid string expression. If *string* contains Null, **Null** is returned.

Remarks

The following example uses the **LTrim**, **RTrim**, and **Trim** functions to trim leading spaces, trailing spaces, and both leading and trailing spaces, respectively:

```
Dim MyVar
MyVar = LTrim("   vbscript ")   ' MyVar contains "vbscript ".
MyVar = RTrim("   vbscript ")   ' MyVar contains "   vbscript".
MyVar = Trim("   vbscript ")   ' MyVar contains "vbscript".
```

Requirements

Version 1

See Also

Left Function | Right Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Math Functions

Abs Function

Atn Function

Cos Function

Exp Function

Fix Function

Int Function

Log Function

Rnd Function

Sgn Function

Sin Function

Sqr Function

Tan Function

[Derived Math Functions]

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Mid Function

Returns a specified number of characters from a string.

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

Arguments

string
: String expression from which characters are returned. If *string* contains Null, **Null** is returned.

start
: Character position in *string* at which the part to be taken begins. If *start* is greater than the number of characters in *string*, **Mid** returns a zero-length string ("").

length
: Number of characters to return. If omitted or if there are fewer than *length* characters in the text (including the character at *start*), all characters from the *start* position to the end of the string are returned.

Remarks

To determine the number of characters in *string*, use the **Len** function.

The following example uses the **Mid** function to return six characters, beginning with the fourth character, in a string:

```
Dim MyVar
MyVar = Mid("VB Script is fun!", 4, 6) ' MyVar contains "Script".
```

Note 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.

Requirements

Version 1

See Also

Left Function | Len Function | LTrim, RTrim, and Trim Functions | Right Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Minute

Returns a whole number between 0 and 59, inclusive, representing the minute of the hour.

`Minute(`*`time`***`)`**

The *time* argument is any expression that can represent a time. If *time* contains Null, **Null** is returned.

Remarks

The following example uses the **Minute** function to return the minute of the hour:

```
Dim MyVar
MyVar = Minute(Now)
```

Requirements

[Version 1]

See Also

[Day Function] | [Hour Function] | [Now Function] | [Second Function] | [Time Function]

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Month Function

Returns a whole number between 1 and 12, inclusive, representing the month of the year.

Month(*date***)**

The *date* argument is any expression that can represent a date. If *date* contains Null, **Null** is returned.

Remarks

The following example uses the **Month** function to return the current month:

```
Dim MyVar
MyVar = Month(Now) ' MyVar contains the number corresponding to
                   ' the current month.
```

Requirements

[Version 1]

See Also

Date Function | Day Function | Now Function | Weekday Function | Year Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

MonthName Function

Returns a string indicating the specified month.

MonthName(*month*[, *abbreviate*]**)**

Arguments

month
: Required. The numeric designation of the month. For example, January is 1, February is 2, and so on.

abbreviate
: Optional. Boolean value that indicates if the month name is to be abbreviated. If omitted, the default is **False**, which means that the month name is not abbreviated.

Remarks

The following example uses the **MonthName** function to return an abbreviated month name for a date expression:

```
Dim MyVar
MyVar = MonthName(10, True) ' MyVar contains "Oct".
```

Requirements

Version 2

See Also

WeekdayName Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

MsgBox Function

Displays a message in a dialog box, waits for the user to click a button, and returns a value indicating which button the user clicked.

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

Arguments

prompt
: String expression displayed as the message in the dialog box. The maximum length of *prompt* is approximately 1024 characters, depending on the width of the characters used. If *prompt* consists of more than one line, you can separate the lines using a carriage return character (**Chr(**13**)**), a linefeed character (**Chr(**10**)**), or carriage return–linefeed character combination (**Chr(**13**) & Chr(**10**)**) between each line.

buttons
: Numeric expression that is the sum of values specifying the number and type of buttons to display, the icon style to use, the identity of the default button, and the modality of the message box. See Settings section for values. If omitted, the default value for *buttons* is 0.

title
: String expression displayed in the title bar of the dialog box. If you omit *title*, the application name is placed in the title bar.

helpfile
: String expression that identifies the Help file to use to provide context-sensitive Help for the dialog box. If *helpfile* is provided, *context* must also be provided. Not available on 16-bit platforms.

context
: Numeric expression that identifies the Help context number assigned by the Help author to the appropriate Help topic. If *context* is

provided, *helpfile* must also be provided. Not available on 16-bit platforms.

Settings

The *buttons* argument settings are:

Constant	Value	Description
vbOKOnly	0	Display **OK** button only.
vbOKCancel	1	Display **OK** and **Cancel** buttons.
vbAbortRetryIgnore	2	Display **Abort**, **Retry**, and **Ignore** buttons.
vbYesNoCancel	3	Display **Yes**, **No**, and **Cancel** buttons.
vbYesNo	4	Display **Yes** and **No** buttons.
vbRetryCancel	5	Display **Retry** and **Cancel** buttons.
vbCritical	16	Display Critical Message icon.
vbQuestion	32	Display Warning Query icon.
vbExclamation	48	Display Warning Message icon.
vbInformation	64	Display Information Message icon.
vbDefaultButton1	0	First button is default.
vbDefaultButton2	256	Second button is default.
vbDefaultButton3	512	Third button is default.
vbDefaultButton4	768	Fourth button is default.
vbApplicationModal	0	Application modal; the user must respond to the message box before continuing work in the current application.
vbSystemModal	4096	System modal; all applications are suspended until the user responds to the message box.

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

Return Values

The **MsgBox** function has the following return values:

Constant	Value	Button
vbOK	1	OK
vbCancel	2	Cancel
vbAbort	3	Abort
vbRetry	4	Retry
vbIgnore	5	Ignore
vbYes	6	Yes
vbNo	7	No

Remarks

When both *helpfile* and *context* are provided, the user can press **F1** to view the Help topic corresponding to the context.

If the dialog box displays a **Cancel** button, pressing the **ESC** key has the same effect as clicking **Cancel**. If the dialog box contains a **Help** button, context-sensitive Help is provided for the dialog box. However, no value is returned until one of the other buttons is clicked.

When the **MsgBox** function is used with Microsoft Internet Explorer, the title of any dialog presented always contains "VBScript:" to differentiate it from standard system dialogs.

The following example uses the **MsgBox** function to display a message box and return a value describing which button was clicked:

```
Dim MyVar
MyVar = MsgBox ("Hello World!", 65, "MsgBox Example")
   ' MyVar contains either 1 or 2, depending on which button is clicked.
```

Requirements

[Version 1](#)

See Also

[InputBox Function](#)

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Now

Returns the current date and time according to the setting of your computer's system date and time.

```
Now
```

Remarks

The following example uses the **Now** function to return the current date and time:

```
Dim MyVar
MyVar = Now ' MyVar contains the current date and time.
```

Requirements

Version 1

See Also

Date Function | Day Function | Hour Function | Minute Function | Month Function | Second Function | Time Function | Weekday Function | Year Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Oct

Returns a string representing the octal value of a number.

`Oct(``number`**`)`**

The *number* argument is any valid expression.

Remarks

If *number* is not already a whole number, it is rounded to the nearest whole number before being evaluated.

If *number* is	Oct returns
Null	Null.
Empty	Zero (0).
Any other number	Up to 11 octal characters,

You can represent octal numbers directly by preceding numbers in the proper range with &O. For example, &O10 is the octal notation for decimal 8.

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

```
Dim MyOct
MyOct = Oct(4)     ' Returns 4.
MyOct = Oct(8)     ' Returns 10.
MyOct = Oct(459)   ' Returns 713.
```

Requirements

Version 1

See Also

Hex Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Replace Function

Returns a string in which a specified substring has been replaced with another substring a specified number of times.

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

Arguments

expression
Required. String expression containing substring to replace.

find
Required. Substring being searched for.

replacewith
Required. Replacement substring.

start
Optional. Position within *expression* where substring search is to begin. If omitted, 1 is assumed. Must be used in conjunction with *count.*

count
Optional. Number of substring substitutions to perform. If omitted, the default value is -1, which means make all possible substitutions. Must be used in conjunction with *start*.

compare
Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. See Settings section for values. If omitted, the default value is 0, which means perform a binary comparison.

Settings

The *compare* argument can have the following values:

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

Return Values

Replace returns the following values:

If	Replace returns
expression is zero-length	Zero-length string ("").
expression is **Null**	An error.
find is zero-length	Copy of expression.
replacewith is zero-length	Copy of *expression* with all occurences of *find* removed.
start > **Len(**expression**)**	Zero-length string.
count is 0	Copy of expression.

Remarks

The return value of the **Replace** function is a string, with substitutions made, that begins at the position specified by *start* and and concludes at the end of the *expression* string. It is not a copy of the original string from start to finish.

The following example uses the **Replace** function to return a string:

```
Dim MyString
MyString = Replace("XXpXXPXXp", "p", "Y")   ' A binary comparison starting at the beginning of the string. Returns
MyString = Replace("XXpXXPXXp", "p", "Y",   ' A textual comparison starting at position 3. Returns "YXXYXXY". 3, -
```

Requirements

[Version 2](#)

See Also

[Filter Function](#)

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

RGB Function

Returns a whole number representing an RGB color value.

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

Arguments

red
: Required. Number in the range 0-255 representing the red component of the color.

green
: Required. Number in the range 0-255 representing the green component of the color.

blue
: Required. Number in the range 0-255 representing the blue component of the color.

Remarks

Application methods and properties that accept a color specification expect that specification to be a number representing an RGB color value. An RGB color value specifies the relative intensity of red, green, and blue to cause a specific color to be displayed.

The low-order byte contains the value for red, the middle byte contains the value for green, and the high-order byte contains the value for blue.

For applications that require the byte order to be reversed, the following function will provide the same information with the bytes reversed:

```
Function RevRGB(red, green, blue)
   RevRGB= CLng(blue + (green * 256) + (red * 65536))
End Function
```

The value for any argument to RGB that exceeds 255 is assumed to be 255.

Requirements

Version 2

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Right Function

Returns a specified number of characters from the right side of a string.

```
Right(string, length)
```

Arguments

string
: String expression from which the rightmost characters are returned. If *string* 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 *string*, the entire string is returned.

Remarks

To determine the number of characters in *string*, use the **Len** function.

The following example uses the **Right** function to return a specified number of characters from the right side of a string:

```
Dim AnyString, MyStr
```

```
AnyString = "Hello World"      ' Define string.
MyStr = Right(AnyString, 1)    ' Returns "d".
MyStr = Right(AnyString, 6)    ' Returns " World".
MyStr = Right(AnyString, 20)   ' Returns "Hello World".
```

> **Note** The **RightB** function is used with byte data contained in a string. Instead of specifying the number of characters to return, *length* specifies the number of bytes.

Requirements

Version 1

See Also

Left Function | Len Function | Mid Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Rnd Function

Returns a random number.

Rnd[**(***number***)**]

The *number* argument can be any valid numeric expression.

Remarks

The **Rnd** function returns a value less than 1 but greater than or equal to 0. The value of *number* determines how **Rnd** generates a random number:

If *number* is	Rnd generates
Less than zero	The same number every time, using *number* as the seed.
Greater than zero	The next random number in the sequence.
Equal to zero	The most recently generated number.
Not supplied	The next random number in the sequence.

For any given initial seed, the same number sequence is generated because each successive call to the **Rnd** function uses the previous number as a seed for the next number in the sequence.

Before calling **Rnd**, use the **Randomize** statement without an argument to initialize the random-number generator with a seed based on the system timer.

To produce random integers in a given range, use this formula:

```
Int((upperbound - lowerbound + 1) * Rnd + lowerbound)
```

Here, *upperbound* is the highest number in the range, and *lowerbound* is the lowest number in the range.

> **Note** To repeat sequences of random numbers, call **Rnd** with a negative argument immediately before using **Randomize** with a numeric argument. Using **Randomize** with the same value for *number* does not repeat the previous sequence.

Requirements

Version 1

See Also

Randomize Statement

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Round Function

Returns a number rounded to a specified number of decimal places.

```
Round(expression[, numdecimalplaces])
```

Arguments

expression
: Required. Numeric expression being rounded.

numdecimalplaces
: 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.

Remarks

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

```
Dim MyVar, pi
pi = 3.14159
MyVar = Round(pi, 2) ' MyVar contains 3.14.
```

Requirements

Version 2

See Also

Int, Fix Functions

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

ScriptEngine Function

Returns a string representing the scripting language in use.

```
ScriptEngine
```

Return Values

The **ScriptEngine** function can return any of the following strings:

String	Description
VBScript	Indicates that Microsoft® Visual Basic® Scripting Edition is the current scripting engine.
JScript	Indicates that Microsoft JScript® is the current scripting engine.
VBA	Indicates that Microsoft Visual Basic for Applications is the current scripting engine.

Remarks

The following example uses the **ScriptEngine** function to return a string describing the scripting language in use:

```
Function GetScriptEngineInfo
   Dim s
   s = ""   ' Build string with necessary info.
   s = ScriptEngine & " Version "
   s = s & ScriptEngineMajorVersion & "."
   s = s & ScriptEngineMinorVersion & "."
   s = s & ScriptEngineBuildVersion
   GetScriptEngineInfo = s   ' Return the results.
End Function
```

Requirements

Version 2

See Also

ScriptEngineBuildVersion Function | ScriptEngineMajorVersion Function | ScriptEngineMinorVersion Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

ScriptEngineBuildVersion Function

Returns the build version number of the scripting engine in use.

```
ScriptEngineBuildVersion
```

Remarks

The return value corresponds directly to the version information contained in the DLL for the scripting language in use.

The following example uses the **ScriptEngineBuildVersion** function to return the build version number of the scripting engine:

```
Function GetScriptEngineInfo
   Dim s
   s = ""   ' Build string with necessary info.
   s = ScriptEngine & " Version "
   s = s & ScriptEngineMajorVersion & "."
   s = s & ScriptEngineMinorVersion & "."
   s = s & ScriptEngineBuildVersion
   GetScriptEngineInfo = s   ' Return the results.
End Function
```

Requirements

Version 2

See Also

ScriptEngine Function | ScriptEngineMajorVersion Function | ScriptEngineMinorVersion Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

ScriptEngineMajorVersion Function

Returns the major version number of the scripting engine in use.

```
ScriptEngineMajorVersion
```

Remarks

The return value corresponds directly to the version information contained in the DLL for the scripting language in use.

The following example uses the **ScriptEngineMajorVersion** function to return the version number of the scripting engine:

```
Function GetScriptEngineInfo
   Dim s
   s = ""   ' Build string with necessary info.
   s = ScriptEngine & " Version "
   s = s & ScriptEngineMajorVersion & "."
   s = s & ScriptEngineMinorVersion & "."
   s = s & ScriptEngineBuildVersion
   GetScriptEngineInfo = s   ' Return the results.
```

```
End Function
```

Requirements

Version 2

See Also

ScriptEngine Function | ScriptEngineBuildVersion Function | ScriptEngineMinorVersion Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

ScriptEngineMinorVersion Function

Returns the minor version number of the scripting engine in use.

```
ScriptEngineMinorVersion
```

Remarks

The return value corresponds directly to the version information contained in the DLL for the scripting language in use.

The following example uses the **ScriptEngineMinorVersion** function to return the minor version number of the scripting engine:

```
Function GetScriptEngineInfo
   Dim s
   s = ""   ' Build string with necessary info.
   s = ScriptEngine & " Version "
   s = s & ScriptEngineMajorVersion & "."
```

```
   s = s & ScriptEngineMinorVersion & "."
   s = s & ScriptEngineBuildVersion
   GetScriptEngineInfo = s   ' Return the results.
End Function
```

Requirements

Version 2

See Also

ScriptEngine Function | ScriptEngineBuildVersion Function | ScriptEngineMajorVersion Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Second Function

Returns a whole number between 0 and 59, inclusive, representing the second of the minute.

`Second(`*`time`*`)`

The *time* argument is any expression that can represent a time. If *time* contains Null, **Null** is returned.

Remarks

The following example uses the **Second** function to return the current second:

```
Dim MySec
MySec = Second(Now)
```

```
' MySec contains the number representing the current second.
```

Requirements

Version 1

See Also

Day Function | Hour Function | Minute Function | Now Function | Time Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

SetLocale Function

Sets the global locale and returns the previous locale.

`SetLocale(`*`lcid`*`)`

The *lcid* argument can be any valid 32-bit value or short string that uniquely identifies a geographical locale. Recognized values can be found in the Locale ID chart.

Remarks

If *lcid* is zero, the locale is set to match the current system setting.

A locale is a set of user preference information related to the user's language, country/region, and cultural conventions. The locale determines such things as keyboard layout, alphabetic sort order, as well as date, time, number, and currency formats.

The following example illustrates the use of the **SetLocale** function. To use this code, paste the entire example between the <BODY> tags of a standard HTML page.

```
Enter Date in UK format: <input type="text" id="UKDate" size="20"><p>
Here's the US equivalent: <input type="text" id="USdate" size="20"><p>
<input type="button" value="Convert" id="button1"><p>
Enter a price in German:   <input type="text" id="GermanNumber" size="20">
<p>
Here's the UK equivalent: <input type="text" id="USNumber" size="20"><p>
<input type="button" value="Convert" id="button2"><p>

<script language="vbscript">
Dim currentLocale
' Get the current locale
currentLocale = GetLocale

Sub Button1_onclick
  Dim original
  original = SetLocale("en-gb")
  mydate = CDate(UKDate.value)
  ' IE always sets the locale to US English so use the
  ' currentLocale variable to set the locale to US English
  original = SetLocale(currentLocale)
  USDate.value = FormatDateTime(mydate,vbShortDate)
End Sub

Sub button2_onclick
  Dim original
  original = SetLocale("de")
  myvalue = CCur(GermanNumber.value)
  original = SetLocale("en-gb")
  USNumber.value = FormatCurrency(myvalue)
End Sub

</script>
```

See Also

GetLocale Function | Locale ID (LCID) Chart

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Sgn Function

Returns an integer indicating the sign of a number.

Sgn(number**)**

The *number* argument can be any valid numeric expression.

Return Values

The **Sgn** function has the following return values:

If number is	Sgn returns
Greater than zero	1
Equal to zero	0
Less than zero	-1

Remarks

The sign of the *number* argument determines the return value of the **Sgn** function.

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

```
Dim MyVar1, MyVar2, MyVar3, MySign
MyVar1 = 12: MyVar2 = -2.4: MyVar3 = 0
MySign = Sgn(MyVar1)   ' Returns 1.
MySign = Sgn(MyVar2)   ' Returns -1.
MySign = Sgn(MyVar3)   ' Returns 0.
```

Requirements

Version 1

See Also

Abs Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Sin Function

Returns the sine of an angle.

`Sin(``number`**`)`**

The *number* argument can be any valid numeric expression that expresses an angle in radians.

Remarks

The **Sin** function takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the hypotenuse. The result lies in the range -1 to 1.

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

The following example uses the **Sin** function to return the sine of an angle:

```
Dim MyAngle, MyCosecant
MyAngle = 1.3   ' Define angle in radians.
MyCosecant = 1 / Sin(MyAngle)   ' Calculate cosecant.
```

Requirements

Version 1

See Also

Atn Function | Cos Function | Derived Math Functions | Tan Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Space Function

Returns a string consisting of the specified number of spaces.

```
Space(number)
```

The *number* argument is the number of spaces you want in the string.

Remarks

The following example uses the **Space** function to return a string consisting of a specified number of spaces:

```
Dim MyString
MyString = Space(10)   ' Returns a string with 10 spaces.
MyString = "Hello" & Space(10) & "World" ' Insert 10 spaces between two strings.
```

Requirements

Version 1

See Also

String Function

Visual Basic Scripting Edition

Split Function

Returns a zero-based, one-dimensional array containing a specified number of substrings.

`Split(``expression[, delimiter[, count[, compare]]]`**`)`**

Arguments

expression
: Required. String expression containing substrings and delimiters. If *expression* is a zero-length string, **Split** returns an empty array, that is, an array with no elements and no data.

delimiter
: Optional. String character used to identify substring limits. If omitted, the space character (" ") is assumed to be the delimiter. If *delimiter* is a zero-length string, a single-element array containing the entire *expression* string is returned.

count
: Optional. Number of substrings to be returned; -1 indicates that all substrings are returned.

compare
: Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. See Settings section for values.

Settings

The *compare* argument can have the following values:

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

Remarks

The following example uses the **Split** function to return an array from a string. The function performs a textual comparison of the delimiter, and returns all of the substrings.

```
Dim MyString, MyArray, Msg
MyString = "VBScriptXisXfun!"
MyArray = Split(MyString, "x", -1, 1)
' MyArray(0) contains "VBScript".
' MyArray(1) contains "is".
' MyArray(2) contains "fun!".
Msg = MyArray(0) & " " & MyArray(1)
Msg = Msg   & " " & MyArray(2)
MsgBox Msg
```

Requirements

Version 2

See Also

Join Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Sqr Function

Returns the square root of a number.

```
Sqr(number)
```

The *number* argument can be any valid numeric expression greater than or equal to 0.

Remarks

The following example uses the **Sqr** function to calculate the square root of a number:

```
Dim MySqr
MySqr = Sqr(4)    ' Returns 2.
MySqr = Sqr(23)    ' Returns 4.79583152331272.
MySqr = Sqr(0)    ' Returns 0.
MySqr = Sqr(-4)    ' Generates a run-time error.
```

Requirements

Version 1

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

StrComp Function

Returns a value indicating the result of a string comparison.

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

Arguments

string1
: Required. Any valid string expression.

string2
: Required. Any valid string expression.

compare
: 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.

Settings

The *compare* argument can have the following values:

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

Return Values

The **StrComp** function has the following return values:

If	StrComp returns
string1 is less than *string2*	-1
string1 is equal to string2	0
string1 is greater than *string2*	1
string1 or *string2* is Null	Null

Remarks

The following example uses the **StrComp** function to return the results of a string comparison. If the third argument is 1, a textual comparison is performed; if the third argument is 0 or omitted, a binary comparison is performed.

```
Dim MyStr1, MyStr2, MyComp
MyStr1 = "ABCD": MyStr2 = "abcd"   ' Define variables.
MyComp = StrComp(MyStr1, MyStr2, 1)   ' Returns 0.
MyComp = StrComp(MyStr1, MyStr2, 0)   ' Returns -1.
MyComp = StrComp(MyStr2, MyStr1)   ' Returns 1.
```

Requirements

Version 1

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

String Function

Returns a repeating character string of the length specified.

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

Arguments

number
: Length of the returned string. If *number* contains Null, **Null** is returned.

character
: Character code specifying the character or string expression whose first character is used to build the return string. If *character* contains **Null**, **Null** is returned.

Remarks

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
```

The following example uses the **String** function to return repeating character strings of the length specified:

```
Dim MyString
```

```
MyString = String(5, "*")   ' Returns "*****".
MyString = String(5, 42)   ' Returns "*****".
MyString = String(10, "ABC")   ' Returns "AAAAAAAAAA".
```

Requirements

Version 1

See Also

Space Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

StrReverse Function

Returns a string in which the character order of a specified string is reversed.

StrReverse(*string1*)

The *string1* argument is the string whose characters are to be reversed. If *string1* is a zero-length string (""), a zero-length string is returned. If *string1* is **Null**, an error occurs.

Remarks

The following example uses the **StrReverse** function to return a string in reverse order:

```
Dim MyStr
MyStr = StrReverse("VBScript") ' MyStr contains "tpircSBV".
```

Requirements

Version 2

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Tan Function

Returns the tangent of an angle.

```
Tan(number)
```

The *number* argument can be any valid numeric expression that expresses an angle in radians.

Remarks

Tan takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle.

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

The following example uses the **Tan** function to return the tangent of an angle:

```
Dim MyAngle, MyCotangent
MyAngle = 1.3   ' Define angle in radians.
MyCotangent = 1 / Tan(MyAngle)   ' Calculate cotangent.
```

Requirements

Version 1

See Also

Atn Function | Cos Function | Derived Math Functions | Sin Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Time Function

Returns a **Variant** of subtype **Date** indicating the current system time.

```
Time
```

Remarks

The following example uses the **Time** function to return the current system time:

```
Dim MyTime
MyTime = Time   ' Return current system time.
```

Requirements

Version 1

See Also

Date Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Timer Function

Returns the number of seconds that have elapsed since 12:00 AM (midnight).

```
Timer
```

Remarks

The following example uses the **Timer** function to determine the time it takes to iterate a **For...Next** loop *N* times:

```
Function TimeIt(N)
   Dim StartTime, EndTime
   StartTime = Timer
   For I = 1 To N
   Next
   EndTime = Timer
   TimeIt = EndTime - StartTime
End Function
```

Requirements

Version 5

See Also

Randomize Statement

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

TimeSerial Function

Returns a **Variant** of subtype **Date** containing the time for a specific hour, minute, and second.

TimeSerial(*hour*, *minute*, *second***)**

Arguments

hour
: Number between 0 (12:00 A.M.) and 23 (11:00 P.M.), inclusive, or a numeric expression.

minute
: Any numeric expression.

second
: Any numeric expression.

Remarks

To specify a time, such as 11:59:59, the range of numbers for each **TimeSerial** argument should be in the accepted range for the unit; that is, 0–23 for hours and 0–59 for minutes and seconds. However, you can also specify relative times for each argument using any numeric expression that represents some number of hours, minutes, or seconds before or after a certain time.

The following example uses expressions instead of absolute time numbers. The **TimeSerial** function returns a time for 15 minutes before (-15) six hours before noon (12 - 6), or 5:45:00 A.M.

```
Dim MyTime1
MyTime1 = TimeSerial(12 - 6, -15, 0) ' Returns 5:45:00 AM.
```

When any argument exceeds the accepted range for that argument, it increments to the next larger unit as appropriate. For example, if you specify 75 minutes, it is evaluated as one hour and 15 minutes. However, if any single argument is outside the range -32,768 to 32,767, or if the time specified by the three arguments, either directly or by expression, causes the date to fall outside the acceptable range of dates, an error occurs.

Requirements

Version 1

See Also

DateSerial Function | DateValue Function | Hour Function | Minute Function | Now Function | Second Function | TimeValue Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

TimeValue

Returns a **Variant** of subtype **Date** containing the time.

```
TimeValue(time)
```

The *time* argument is usually a string expression representing a time from 0:00:00 (12:00:00 A.M.) to 23:59:59 (11:59:59 P.M.), inclusive. However, *time* can also be any expression that represents a time in that range. If *time* contains Null, **Null** is returned.

Remarks

You can enter valid times using a 12-hour or 24-hour clock. For example, "2:24PM" and "14:24" are both valid *time* arguments. If the *time* argument contains date information, **TimeValue** doesn't return the date information. However, if *time* includes invalid date information, an error occurs.

The following example uses the **TimeValue** function to convert a string to a time. You can also use date literals to directly assign a time to a **Variant** (for example, MyTime = #4:35:17 PM#).

```
Dim MyTime
MyTime = TimeValue("4:35:17 PM")   ' MyTime contains 4:35:17 PM.
```

Requirements

Version 1

See Also

DateSerial Function | DateValue Function | Hour Function | Minute Function | Now Function | Second Function | TimeSerial Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

TypeName Function

Returns a string that provides **Variant** subtype information about a variable.

```
TypeName(varname)
```

The required *varname* argument can be any variable.

Return Values

The **TypeName** function has the following return values:

Value	Description

Byte	Byte value
Integer	Integer value
Long	Long integer value
Single	Single-precision floating-point value
Double	Double-precision floating-point value
Currency	Currency value
Decimal	Decimal value
Date	Date or time value
String	Character string value
Boolean	Boolean value; **True** or **False**
Empty	Unitialized
Null	No valid data
<object type>	Actual type name of an object
Object	Generic object
Unknown	Unknown object type
Nothing	Object variable that doesn't yet refer to an object instance
Error	Error

Remarks

The following example uses the **TypeName** function to return information about a variable:

```
Dim ArrayVar(4), MyType
NullVar = Null   ' Assign Null value.

MyType = TypeName("VBScript")   ' Returns "String".
MyType = TypeName(4)            ' Returns "Integer".
MyType = TypeName(37.50)        ' Returns "Double".
MyType = TypeName(NullVar)      ' Returns "Null".
MyType = TypeName(ArrayVar)     ' Returns "Variant()".
```

Requirements

Version 2

See Also

[IsArray Function](#) | [IsDate Function](#) | [IsEmpty Function](#) | [IsNull Function](#) | [IsNumeric Function](#) | [IsObject Function](#) | [VarType Function](#)

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

UBound Function

Returns the largest available subscript for the indicated dimension of an array.

`UBound(``arrayname[, dimension]`**`)`**

Arguments

arrayname
    Required. Name of the array variable; follows standard variable naming conventions.
dimension
    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.

Remarks

The **UBound** function is used with the **LBound** function to determine the size of an array. Use the **LBound** function to find the lower limit of an array dimension.

The lower bound for any dimension is always 0. As a result, **UBound** returns the following values for an array with these dimensions:

```
Dim A(100,3,4)
```

Statement	Return Value

UBound(A, 1)	100
UBound(A, 2)	3
UBound(A, 3)	4

Requirements

Version 1

See Also

Dim Statement | LBound Function | ReDim Statement

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

UCase Function

Returns a string that has been converted to uppercase.

`UCase(`*`string`*`)`

The *string* argument is any valid string expression. If *string* contains Null, **Null** is returned.

Remarks

Only lowercase letters are converted to uppercase; all uppercase letters and non-letter characters remain unchanged.

The following example uses the **UCase** function to return an uppercase version of a string:

```
Dim MyWord
MyWord = UCase("Hello World")   ' Returns "HELLO WORLD".
```

Requirements

Version 1

See Also

LCase Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

VarType Function

Returns a value indicating the subtype of a variable.

```
VarType(varname)
```

The *varname* argument can be any variable.

Return Values

The **VarType** function returns the following values:

Constant	Value	Description
vbEmpty	0	Empty (uninitialized)
vbNull	1	Null (no valid data)

vbInteger	2	Integer
vbLong	3	Long integer
vbSingle	4	Single-precision floating-point number
vbDouble	5	Double-precision floating-point number
vbCurrency	6	Currency
vbDate	7	Date
vbString	8	String
vbObject	9	Automation object
vbError	10	Error
vbBoolean	11	Boolean
vbVariant	12	Variant (used only with arrays of Variants)
vbDataObject	13	A data-access object
vbByte	17	Byte
vbArray	8192	Array

> **Note** These constants are specified by VBScript. As a result, the names can be used anywhere in your code in place of the actual values.

Remarks

The **VarType** function never returns the value for Array by itself. It is always added to some other value to indicate an array of a particular type. The value for Variant is only returned when it has been added to the value for Array to indicate that the argument to the **VarType** function is an array. For example, the value returned for an array of integers is calculated as 2 + 8192, or 8194. If an object has a default property, **VarType (***object***)** returns the type of its default property.

The following example uses the **VarType** function to determine the subtype of a variable.

```
Dim MyCheck
MyCheck = VarType(300)          ' Returns 2.
MyCheck = VarType(#10/19/62#)   ' Returns 7.
MyCheck = VarType("VBScript")   ' Returns 8.
```

Requirements

Version 1

See Also

[IsArray Function] | [IsDate Function] | [IsEmpty Function] | [IsNull Function] | [IsNumeric Function] | [IsObject Function] | [TypeName Function]

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Weekday Function

Returns a whole number representing the day of the week.

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

Arguments

date
Any expression that can represent a date. If *date* contains Null, **Null** is returned.
firstdayofweek
A constant that specifies the first day of the week. If omitted, **vbSunday** is assumed.

Settings

The *firstdayofweek* argument has these settings:

Constant	Value	Description
vbUseSystemDayOfWeek	0	Use National Language Support (NLS) API setting.
vbSunday	1	Sunday
vbMonday	2	Monday
vbTuesday	3	Tuesday

vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

Return Values

The **Weekday** function can return any of these values:

Constant	Value	Description
vbSunday	1	Sunday
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

Remarks

The following example uses the **Weekday** function to obtain the day of the week from a specified date:

```
Dim MyDate, MyWeekDay
MyDate = #October 19, 1962#   ' Assign a date.
MyWeekDay = Weekday(MyDate)   ' MyWeekDay contains 6 because MyDate represents a Friday.
```

Requirements

Version 1

See Also

Date Function | Day Function | Month Function | Now Function | Year Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

WeekdayName Function

Returns a string indicating the specified day of the week.

`WeekdayName(``weekday, abbreviate, firstdayofweek`**`)`**

Arguments

weekday
Required. The numeric designation for the day of the week. Numeric value of each day depends on setting of the *firstdayofweek* setting.

abbreviate
Optional. Boolean value that indicates if the weekday name is to be abbreviated. If omitted, the default is **False**, which means that the weekday name is not abbreviated.

firstdayofweek
Optional. Numeric value indicating the first day of the week. See Settings section for values.

Settings

The *firstdayofweek* argument can have the following values:

Constant	Value	Description
vbUseSystemDayOfWeek	0	Use National Language Support (NLS) API setting.
vbSunday	1	Sunday (default)
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

Remarks

The following example uses the **WeekDayName** function to return the specified day:

```
Dim MyDate
MyDate = WeekDayName(6, True)   ' MyDate contains Fri.
```

Requirements

Version 2

See Also

MonthName Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Year Function

Returns a whole number representing the year.

```
Year(date)
```

The *date* argument is any expression that can represent a date. If *date* contains Null, **Null** is returned.

Remarks

The following example uses the **Year** function to obtain the year from a specified date:

```
Dim MyDate, MyYear
MyDate = #October 19, 1962#   ' Assign a date.
MyYear = Year(MyDate)         ' MyYear contains 1962.
```

Requirements

Version 1

See Also

Date Function | Day Function | Month Function | Now Function | Weekday Function

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

VBScript Keywords

Empty

False

Nothing

Null

True

Related Sections

VBScript Langauge Reference

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Empty

The **Empty** keyword is used to indicate an uninitialized variable value. This is not the same thing as **Null**.

See Also

Null

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

False

The **False** keyword has a value equal to 0.

See Also

True

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Nothing

The **Nothing** keyword in VBScript is used to disassociate an object variable from any actual object. Use the **Set** statement to assign **Nothing** to an object variable. For example:

```
Set MyObject = Nothing
```

Several object variables can refer to the same actual object. When **Nothing** is assigned to an object variable, that variable no longer refers to any actual object. When several object variables refer to the same object, memory and system resources associated with the object to which the variables refer are released only after all of them have been set to **Nothing**, either explicitly using **Set**, or implicitly after the last object variable set to **Nothing** goes out of scope.

See Also

Dim Statement | Set Statement

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Null

The **Null** keyword is used to indicate that a variable contains no valid data. This is not the same thing as **Empty**.

See Also

Empty

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

True

The **True** keyword has a value equal to -1.

See Also

False

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Methods

In This Section

Related Sections

VBScript Langauge Reference

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Clear Method

Clears all property settings of the **Err** object.

object.**Clear**

The object is always the **Err** object.

Remarks

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

- On Error Resume Next
- Exit Sub
- Exit Function

The following example illustrates use of the **Clear** method.

```
On Error Resume Next
Err.Raise 6   ' Raise an overflow error.
MsgBox ("Error # " & CStr(Err.Number) & " " & Err.Description)
Err.Clear     ' Clear the error.
```

Requirements

Version 1

See Also

Description Property | Err Object | Number Property | OnError Statement | Raise Method | Source Property

Applies To: Err Object

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Execute Method

Executes a regular expression search against a specified string.

```
object.Execute(string)
```

Arguments

object
Required. Always the name of a **RegExp** object.

string
Required. The text string upon which the regular expression is executed.

Remarks

The actual pattern for the regular expression search is set using the **Pattern** property of the **RegExp** object.

The **Execute** method returns a **Matches** collection containing a **Match** object for each match found in *string*. **Execute** returns an empty **Matches** collection if no match is found.

The following code illustrates the use of the **Execute** method.

```
Function RegExpTest(patrn, strng)
  Dim regEx, Match, Matches      ' Create variable.
  Set regEx = New RegExp         ' Create a regular expression.
  regEx.Pattern = patrn         ' Set pattern.
  regEx.IgnoreCase = True         ' Set case insensitivity.
  regEx.Global = True         ' Set global applicability.
  Set Matches = regEx.Execute(strng)   ' Execute search.
  For Each Match in Matches      ' Iterate Matches collection.
    RetStr = RetStr & "Match found at position "
    RetStr = RetStr & Match.FirstIndex & ". Match Value is '"
    RetStr = RetStr & Match.Value & "'." & vbCRLF
  Next
  RegExpTest = RetStr
End Function
MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
```

Requirements

Version 5

See Also

Replace Method | Test Method

Applies To: RegExp Object

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Raise Method

Generates a run-time error.

`object.`**`Raise(`**`number, source, description, helpfile, helpcontext`**`)`**

Arguments

object
: Always the **Err** object.

number
: A **Long** integer subtype that identifies the nature of the error. VBScript errors (both VBScript-defined and user-defined errors) are in the range 0–65535.

source
: A string expression naming the object or application that originally generated the error. When setting this property for an Automation object, use the form *project***.***class*. If nothing is specified, the programmatic ID of the current VBScript project is used.

description
: A string expression describing the error. If unspecified, the value in *number* is examined. If it can be mapped to a VBScript run-time error code, a string provided by VBScript is used as *description.* If there is no VBScript error corresponding to *number*, a generic error message is used.

helpfile

The fully qualified path to the Help file in which help on this error can be found. If unspecified, VBScript uses the fully qualified drive, path, and file name of the VBScript Help file.

helpcontext

The context ID identifying a topic within *helpfile* that provides help for the error. If omitted, the VBScript Help file context ID for the error corresponding to the *number* property is used, if it exists.

Remarks

All the arguments are optional except *number*. If you use **Raise**, however, without specifying some arguments, and the property settings of the **Err** object contain values that have not been cleared, those values become the values for your error.

When setting the *number* property to your own error code in an Automation object, you add your error code number to the constant **vbObjectError**. For example, to generate the error number 1050, assign **vbObjectError** + 1050 to the *number* property.

The following example illustrates use of the **Raise** method.

```
On Error Resume Next
Err.Raise 6  ' Raise an overflow error.
MsgBox ("Error # " & CStr(Err.Number) & " " & Err.Description)
Err.Clear    ' Clear the error.
```

Requirements

Version 1

See Also

Clear Method | Description Property | Err Object | Number Property Source Property

Applies To: Err Object

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Replace Method

Replaces text found in a regular expression search.

`object.Replace(string1, string2)`

Arguments

object
: Required. Always the name of a **RegExp** object.

string1
: Required. *String1* is the text string in which the text replacement is to occur.

string2
: Required. *String2* is the replacement text string.

Remarks

The actual pattern for the text being replaced is set using the **Pattern** property of the **RegExp** object.

The **Replace** method returns a copy of *string1* with the text of **RegExp.Pattern** replaced with *string2*. If no match is found, a copy of *string1* is returned unchanged.

The following code illustrates use of the **Replace** method.

```
Function ReplaceTest(patrn, replStr)
  Dim regEx, str1               ' Create variables.
  str1 = "The quick brown fox jumped over the lazy dog."
  Set regEx = New RegExp            ' Create regular expression.
  regEx.Pattern = patrn            ' Set pattern.
  regEx.IgnoreCase = True            ' Make case insensitive.
  ReplaceTest = regEx.Replace(str1, replStr)   ' Make replacement.
End Function

MsgBox(ReplaceTest("fox", "cat"))      ' Replace 'fox' with 'cat'.
```

In addition, the **Replace** method can replace subexpressions in the pattern. The following call to the function shown in the previous example swaps each pair of words in the original string:

```
MsgBox(ReplaceText("(\S+)(\s+)(\S+)", "$3$2$1"))   ' Swap pairs of words.
```

Requirements

Version 5

See Also

Execute Method | Test Method

Applies To: RegExp Object

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Test Method

Executes a regular expression search against a specified string and returns a **Boolean** value that indicates if a pattern match was found.

```
object.Test(string)
```

Arguments

object
Required. Always the name of a **RegExp** object.

string
Required. The text string upon which the regular expression is executed.

Remarks

The actual pattern for the regular expression search is set using the **Pattern** property of the **RegExp** object. The **RegExp.Global** property has no effect on the **Test** method.

The **Test** method returns **True** if a pattern match is found; **False** if no match is found.

The following code illustrates the use of the **Test** method.

```
Function RegExpTest(patrn, strng)
  Dim regEx, retVal            ' Create variable.
  Set regEx = New RegExp         ' Create regular expression.
  regEx.Pattern = patrn         ' Set pattern.
  regEx.IgnoreCase = False      ' Set case sensitivity.
  retVal = regEx.Test(strng)      ' Execute the search test.
  If retVal Then
    RegExpTest = "One or more matches were found."
  Else
    RegExpTest = "No match was found."
  End If
End Function
MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
```

Requirements

Version 5

See Also

Execute Method | Replace Method

Applies To: RegExp Object

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Miscellaneous

In This Section

Related Sections

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Character Set (0 - 127)

The following table lists 0 - 127.

Code	Char	Code	Char	Code	Char	Code	Char
0		32	[space]	64	@	96	`
1		33	!	65	A	97	a
2		34	"	66	B	98	b
3		35	#	67	C	99	c

4		36	$	68	D	100	d
5		37	%	69	E	101	e
6		38	&	70	F	102	f
7		39	'	71	G	103	g
8	**	40	(	72	H	104	h
9	**	41	)	73	I	105	i
10	**	42	*	74	J	106	j
11		43	+	75	K	107	k
12		44	,	76	L	108	l
13	**	45	-	77	M	109	m
14		46	.	78	N	110	n
15		47	/	79	O	111	o
16		48	0	80	P	112	p
17		49	1	81	Q	113	q
18		50	2	82	R	114	r
19		51	3	83	S	115	s
20		52	4	84	T	116	t
21		53	5	85	U	117	u
22		54	6	86	V	118	v
23		55	7	87	W	119	w
24		56	8	88	X	120	x
25		57	9	89	Y	121	y
26		58	:	90	Z	122	z
27		59	;	91	[	123	{
28		60	<	92	\	124	|
29		61	=	93	]	125	}
30	-	62	>	94	^	126	~
31		63	?	95	_	127	•

** Values 8, 9, 10, and 13 convert to backspace, tab, linefeed, and carriage return characters, respectively. They have no graphical representation, but depending on the application, may affect the visual display of text.

- Not supported on the current platform.

See Also

Character Set (128 - 255)

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Character Set (128 - 255)

The following table lists 128 - 255.

Code	Char	Code	Char	Code	Char	Code	Char
128	€	160	[space]	192	À	224	à
129	•	161	¡	193	Á	225	á
130	‚	162	¢	194	Â	226	â
131	ƒ	163	£	195	Ã	227	ã
132	„	164	¤	196	Ä	228	ä
133	…	165	¥	197	Å	229	å
134	†	166	¦	198	Æ	230	æ
135	‡	167	§	199	Ç	231	ç
136	ˆ	168	¨	200	È	231	ç
137	‰	169	©	201	É	232	è
138	Š	170	ª	202	Ê	233	é
139	‹	171	«	203	Ë	234	ê
140	Œ	172	¬	204	Ì	235	ë
141	•	173		205	Í	236	ì
142	Ž	174	®	206	Î	237	í
143	•	175	¯	207	Ï	238	î

144	•	176	°	208	Ð	239	ï
145	'	177	±	209	Ñ	240	ð
146	'	178	²	210	Ò	241	ñ
147	"	179	³	211	Ó	242	ò
148	"	180	´	212	Ô	243	ó
149	·	181	µ	213	Õ	244	ô
150	–	182	¶	214	Ö	245	õ
151	—	183	·	215	×	246	ö
152	˜	184	¸	216	Ø	247	÷
153	™	185	¹	217	Ù	248	ø
154	š	186	º	218	Ú	249	ù
155	›	187	››	219	Û	250	ú
156	œ	188	¼	220	Ü	251	û
157	•	189	½	221	Ý	252	ü
158	ž	190	¾	222	Þ	253	ý
159	Ÿ	191	¿	223	ß	254	þ

- Not supported on the current platform.

See Also

Character Set (0 - 127)

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Locale ID (LCID) Chart

The following table lists Locale IDs (LCID).

Locale Description	Short String	Hex Value	Decimal Value	Locale Description	Short String	Hex Value	Decimal Value
Afrikaans	af	0x0436	1078	Icelandic	is	0x040F	1039
Albanian	sq	0x041C	1052	Indonesian	id	0x0421	1057
Arabic – United Arab Emirates	ar-ae	0x3801	14337	Italian - Italy	it-it	0x0410	1040
Arabic - Bahrain	ar-bh	0x3C01	15361	Italian - Switzerland	it-ch	0x0810	2064
Arabic - Algeria	ar-dz	0x1401	5121	Japanese	ja	0x0411	1041
Arabic - Egypt	ar-eg	0x0C01	3073	Korean	ko	0x0412	1042
Arabic - Iraq	ar-iq	0x0801	2049	Latvian	lv	0x0426	1062
Arabic - Jordan	ar-jo	0x2C01	11265	Lithuanian	lt	0x0427	1063
Arabic - Kuwait	ar-kw	0x3401	13313	FYRO Macedonian	mk	0x042F	1071
Arabic - Lebanon	ar-lb	0x3001	12289	Malay - Malaysia	ms-my	0x043E	1086
Arabic - Libya	ar-ly	0x1001	4097	Malay – Brunei	ms-bn	0x083E	2110
Arabic - Morocco	ar-ma	0x1801	6145	Maltese	mt	0x043A	1082
Arabic - Oman	ar-om	0x2001	8193	Marathi	mr	0x044E	1102
Arabic - Qatar	ar-qa	0x4001	16385	Norwegian - Bokmål	no-no	0x0414	1044
Arabic - Saudi Arabia	ar-sa	0x0401	1025	Norwegian – Nynorsk	no-no	0x0814	2068
Arabic - Syria	ar-sy	0x2801	10241	Polish	pl	0x0415	1045
Arabic - Tunisia	ar-tn	0x1C01	7169	Portuguese - Portugal	pt-pt	0x0816	2070
Arabic - Yemen	ar-ye	0x2401	9217	Portuguese - Brazil	pt-br	0x0416	1046
Armenian	hy	0x042B	1067	Raeto-Romance	rm	0x0417	1047
Azeri – Latin	az-az	0x042C	1068	Romanian - Romania	ro	0x0418	1048
Azeri – Cyrillic	az-az	0x082C	2092	Romanian - Moldova	ro-mo	0x0818	2072
Basque	eu	0x042D	1069	Russian	ru	0x0419	1049
Belarusian	be	0x0423	1059	Russian - Moldova	ru-mo	0x0819	2073
Bulgarian	bg	0x0402	1026	Sanskrit	sa	0x044F	1103
Catalan	ca	0x0403	1027	Serbian - Cyrillic	sr-sp	0x0C1A	3098
Chinese - China	zh-cn	0x0804	2052	Serbian – Latin	sr-sp	0x081A	2074
Chinese - Hong Kong S.A.R.	zh-hk	0x0C04	3076	Setsuana	tn	0x0432	1074
Chinese – Macau S.A.R	zh-mo	0x1404	5124	Slovenian	sl	0x0424	1060

Chinese - Singapore	zh-sg	0x1004	4100	Slovak	sk	0x041B	1051
Chinese - Taiwan	zh-tw	0x0404	1028	Sorbian	sb	0x042E	1070
Croatian	hr	0x041A	1050	Spanish - Spain	es-es	0x0C0A	1034
Czech	cs	0x0405	1029	Spanish - Argentina	es-ar	0x2C0A	11274
Danish	da	0x0406	1030	Spanish - Bolivia	es-bo	0x400A	16394
Dutch – The Netherlands	nl-nl	0x0413	1043	Spanish - Chile	es-cl	0x340A	13322
Dutch - Belgium	nl-be	0x0813	2067	Spanish - Colombia	es-co	0x240A	9226
English - Australia	en-au	0x0C09	3081	Spanish - Costa Rica	es-cr	0x140A	5130
English - Belize	en-bz	0x2809	10249	Spanish - Dominican Republic	es-do	0x1C0A	7178
English - Canada	en-ca	0x1009	4105	Spanish - Ecuador	es-ec	0x300A	12298
English – Carribbean	en-cb	0x2409	9225	Spanish - Guatemala	es-gt	0x100A	4106
English - Ireland	en-ie	0x1809	6153	Spanish - Honduras	es-hn	0x480A	18442
English - Jamaica	en-jm	0x2009	8201	Spanish - Mexico	es-mx	0x080A	2058
English - New Zealand	en-nz	0x1409	5129	Spanish - Nicaragua	es-ni	0x4C0A	19466
English – Phillippines	en-ph	0x3409	13321	Spanish - Panama	es-pa	0x180A	6154
English - South Africa	en-za	0x1C09	7177	Spanish - Peru	es-pe	0x280A	10250
English - Trinidad	en-tt	0x2C09	11273	Spanish - Puerto Rico	es-pr	0x500A	20490
English - United Kingdom	en-gb	0x0809	2057	Spanish - Paraguay	es-py	0x3C0A	15370
English - United States	en-us	0x0409	1033	Spanish - El Salvador	es-sv	0x440A	17418
Estonian	et	0x0425	1061	Spanish - Uruguay	es-uy	0x380A	14346
Farsi	fa	0x0429	1065	Spanish - Venezuela	es-ve	0x200A	8202
Finnish	fi	0x040B	1035	Sutu	sx	0x0430	1072
Faroese	fo	0x0438	1080	Swahili	sw	0x0441	1089
French - France	fr-fr	0x040C	1036	Swedish - Sweden	sv-se	0x041D	1053
French - Belgium	fr-be	0x080C	2060	Swedish - Finland	sv-fi	0x081D	2077
French - Canada	fr-ca	0x0C0C	3084	Tamil	ta	0x0449	1097
French - Luxembourg	fr-lu	0x140C	5132	Tatar	tt	0X0444	1092
French - Switzerland	fr-ch	0x100C	4108	Thai	th	0x041E	1054

Gaelic – Ireland	gd-ie	0x083C	2108	Turkish	tr	0x041F	1055
Gaelic - Scotland	gd	0x043C	1084	Tsonga	ts	0x0431	1073
German - Germany	de-de	0x0407	1031	Ukrainian	uk	0x0422	1058
German - Austria	de-at	0x0C07	3079	Urdu	ur	0x0420	1056
German - Liechtenstein	de-li	0x1407	5127	Uzbek – Cyrillic	uz-uz	0x0843	2115
German - Luxembourg	de-lu	0x1007	4103	Uzbek – Latin	uz-uz	0x0443	1091
German - Switzerland	de-ch	0x0807	2055	Vietnamese	vi	0x042A	1066
Greek	el	0x0408	1032	Xhosa	xh	0x0434	1076
Hebrew	he	0x040D	1037	Yiddish	yi	0x043D	1085
Hindi	hi	0x0439	1081	Zulu	zu	0x0435	1077
Hungarian	hu	0x040E	1038				

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Objects and Collections

In This Section

Class Object

Err Object

Matches Collection

Match Object

Regular Expression (RegExp) Object

SubMatches Collection

Related Sections

VBScript Langauge Reference

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Class Object

The object created using the **Class** statement. Provides access to the events of the class.

Remarks

You cannot explicitly declare a variable to be of type Class. In the VBScript context, the term "class object" refers to any object defined using the VBScript **Class** statement.

Once you have created a class definition using the **Class** statement, you can create an instance of the class using the following form:

```
Dim X
Set X = New classname
```

Because VBScript is a late-bound language, you cannot do any of the following:

```
Dim X as New classname
```

-or-

```
Dim X
X = New classname
```

-or-

```
Set X = New Scripting.FileSystemObject
```

Events

Class Object Events

Requirements

Version 5

See Also

Class Statement | Dim Statement | Set Statement

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Class Object Events

The Class object provides access to the events of the class.

Events

Initialize Event

[Terminate Event]

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Matches Collection

Collection of regular expression **Match** objects.

Remarks

A **Matches** collection contains individual **Match** objects, and can be only created using the **Execute** method of the **RegExp** object. The **Matches** collection's one property is read-only, as are the individual **Match** object properties.

When a regular expression is executed, zero or more **Match** objects can result. Each **Match** object provides access to the string found by the regular expression, the length of the string, and an index to where the match was found.

The following code illustrates how to obtain a **Matches** collection from a regular expression search and how to iterate the collection:

```
Function RegExpTest(patrn, strng)
   Dim regEx, Match, Matches   ' Create variable.
   Set regEx = New RegExp   ' Create regular expression.
   regEx.Pattern = patrn   ' Set pattern.
   regEx.IgnoreCase = True   ' Set case insensitivity.
   regEx.Global = True   ' Set global applicability.
   Set Matches = regEx.Execute(strng)   ' Execute search.
   For Each Match in Matches   ' Iterate Matches collection.
      RetStr = RetStr & "Match found at position "
      RetStr = RetStr & Match.FirstIndex & ". Match Value is '"
      RetStr = RetStr & Match.Value & "'." & vbCRLF
   Next
   RegExpTest = RetStr
```

```
End Function
MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
```

Requirements

Version 1

See Also

For Each...Next Statement | Match Object | RegExp Object | SubMatches Collection

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Err Object

Contains information about run-time errors. Accepts the **Raise** and **Clear** methods for generating and clearing run-time errors.

Remarks

The **Err** object is an intrinsic object with global scope — there is no need to create an instance of it in your code. The properties of the **Err** object are set by the generator of an error — Visual Basic, an Automation object, or the VBScript programmer.

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

When a run-time error occurs, the properties of the **Err** object are filled with information that uniquely identifies the error and information that can be used to handle it. To generate a run-time error in your code, use the **Raise** method.

The **Err** object's properties are reset to zero or zero-length strings ("") after an **On Error Resume Next** statement. The **Clear** method can be used to explicitly reset **Err**.

The following example illustrates use of the **Err** object:

```
On Error Resume Next
Err.Raise 6   ' Raise an overflow error.
MsgBox ("Error # " & CStr(Err.Number) & " " & Err.Description)
Err.Clear      ' Clear the error.
```

Properties and Methods

Err Object Properties and Methods

Requirements

Version 1

See Also

On Error Statement

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Err Object Properties and Methods

The Err object contains information about run-time errors.

Properties

Description Property

HelpContext Property

HelpFile Property

Number Property

Source Property

Methods

Clear Method

Raise Method

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Match Object

Provides access to the read-only properties of a regular expression match.

Remarks

A **Match** object can be only created using the **Execute** method of the **RegExp** object, which actually returns a collection of **Match** objects. All **Match** object properties are read-only.

When a regular expression is executed, zero or more **Match** objects can result. Each **Match** object provides access to the string found by the regular expression, the length of the string, and an index to where the match was found.

The following code illustrates the use of the **Match** object:

```
Function RegExpTest(patrn, strng)
   Dim regEx, Match, Matches   ' Create variable.
   Set regEx = New RegExp   ' Create regular expression.
   regEx.Pattern = patrn   ' Set pattern.
   regEx.IgnoreCase = True   ' Set case insensitivity.
   regEx.Global = True   ' Set global applicability.
   Set Matches = regEx.Execute(strng)   ' Execute search.
   For Each Match in Matches   ' Iterate Matches collection.
      RetStr = RetStr & "Match " & I & " found at position "
      RetStr = RetStr & Match.FirstIndex & ". Match Value is "'
      RetStr = RetStr & Match.Value & "'." & vbCRLF
   Next
   RegExpTest = RetStr
End Function
MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
```

Properties

Match Object Properties

Requirements

Version 5

See Also

Matches Collection | RegExp Object | SubMatches Colleciton

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Match Object Properties

The Match object provides access to the read-only properties of a regular expression match.

Properties

FirstIndex Property

Length Property

Value Property

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Regular Expression (RegExp) Object

Provides simple regular expression support.

Remarks

The following code illustrates the use of the **RegExp** object.

```
Function RegExpTest(patrn, strng)
   Dim regEx, Match, Matches   ' Create variable.
   Set regEx = New RegExp   ' Create a regular expression.
   regEx.Pattern = patrn   ' Set pattern.
```

```
   regEx.IgnoreCase = True   ' Set case insensitivity.
   regEx.Global = True   ' Set global applicability.
   Set Matches = regEx.Execute(strng)   ' Execute search.
   For Each Match in Matches   ' Iterate Matches collection.
      RetStr = RetStr & "Match found at position "
      RetStr = RetStr & Match.FirstIndex & ". Match Value is '"
      RetStr = RetStr & Match.Value & "'." & vbCRLF
   Next
   RegExpTest = RetStr
End Function
MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
```

Properties and Methods

Regular Expression Object Properties and Methods

Requirements

Version 5

See Also

Match Object | Matches Collection

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Regular Expression Object Properties and Methods

The Regular Expression object provides simple regular expression support.

Properties

Global Property

IgnoreCase Property

Pattern Property

Methods

Execute Method

Replace Method

Test Method

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

SubMatches Collection

Collection of regular expression submatch strings.

Remarks

A **SubMatches** collection contains individual submatch strings, and can only be created using the **Execute** method of the **RegExp** object. The **SubMatches** collection's properties are read-only

When a regular expression is executed, zero or more submatches can result when subexpressions are enclosed in capturing parentheses. Each item in the **SubMatches** collection is the string found and captured by the regular expression.

The following code illustrates how to obtain a **SubMatches** collection from a regular expression search and how to access its individual members:

```
Function SubMatchTest(inpStr)
  Dim oRe, oMatch, oMatches
  Set oRe = New RegExp
  ' Look for an e-mail address (not a perfect RegExp)
  oRe.Pattern = "(\w+)@(\w+)\.(\w+)"
  ' Get the Matches collection
  Set oMatches = oRe.Execute(inpStr)
  ' Get the first item in the Matches collection
  Set oMatch = oMatches(0)
  ' Create the results string.
  ' The Match object is the entire match - dragon@xyzzy.com
  retStr = "Email address is: " & oMatch & vbNewline
  ' Get the sub-matched parts of the address.
  retStr = retStr & "Email alias is: " & oMatch.SubMatches(0)  ' dragon
  retStr = retStr & vbNewline
  retStr = retStr & "Organization is: " & oMatch. SubMatches(1)' xyzzy
  SubMatchTest = retStr
End Function

MsgBox(SubMatchTest("Please send mail to dragon@xyzzy.com. Thanks!"))
```

Requirements

Version 5.5

See Also

For Each...Next Statement | Match Object | Matches Collection |RegExp Object

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Operators

In This Section

Operator Precedence

Operator Summary

Arithmetic Operators

Comparison Operators

Concatenation Operators

Logical Operators

Related Sections

VBScript Langauge Reference

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Operator Precedence

When several operations occur in an expression, each part is evaluated and resolved in a predetermined order called operator precedence.

Parentheses can be used to override the order of precedence and force some parts of an expression to be evaluated before other parts. Operations within parentheses are always performed before those outside. Within parentheses, however, normal operator precedence is maintained.

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

Arithmetic	Comparison	Logical
Negation (**-**)	Equality (=)	Not
Exponentiation (^)	Inequality (<>)	And
Multiplication and division (*****, **/**)	Less than (<)	Or
Integer division (\)	Greater than (>)	Xor
Modulus arithmetic (**Mod**)	Less than or equal to (<=)	Eqv
Addition and subtraction (+, **-**)	Greater than or equal to (>=)	Imp
String concatenation (**&**)	Is	&

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

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

Requirements

Version 1

See Also

Is Operator | Operator Summary

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Operator Summary

Arithmetic Operators
Operators used to perform mathematical calculations.

Assignment Operator
Operator used to assign a value to a property or variable.

Comparison Operators
Operators used to perform comparisons.

Concatenation Operators
Operators used to combine strings.

Logical Operators
Operators used to perform logical operations.

See Also

Operator Precedence

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Addition Operator (+)

Sums two numbers.

```
result = expression1 + expression2
```

Arguments

result
Any numeric variable.
expression1
Any expression.
expression2
Any expression.

Remarks

Although you can also use the **+** operator to concatenate two character strings, you should use the **&** operator for concatenation to eliminate ambiguity and provide self-documenting code.

When you use the **+** operator, you may not be able to determine whether addition or string concatenation will occur.

The underlying subtype of the expressions determines the behavior of the **+** operator in the following way:

If	Then
Both expressions are numeric	Add.
Both expressions are strings	Concatenate.
One expression is numeric and the other is a string	Add.

If one or both expressions are Null expressions, *result* is **Null**. If both expressions are Empty, *result* is an **Integer** subtype. However, if only one expression is **Empty**, the other expression is returned unchanged as *result.*

Requirements

Version 1

See Also

& Operator | - Operator | Arithmetic Operators | Concatenation Operators | Operator Precedence | Operator Summary

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

And Operator

Performs a logical conjunction on two expressions.

```
result = expression1 And expression2
```

Arguments

result
Any numeric variable.

expression1
Any expression.

expression2
Any expression.

Remarks

If, and only if, both expressions evaluate to **True**, *result* is **True**. If either expression evaluates to **False**, *result* is **False**. The following table illustrates how *result* is determined:

If expression1 is	And expression2 is	The result is
True	True	True

True	False	False
True	Null	Null
False	True	False
False	False	False
False	Null	False
Null	True	Null
Null	False	False
Null	Null	Null

The **And** operator also performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following table:

If bit in *expression1* is	And bit in *expression2* is	The *result* is
0	0	0
0	1	0
1	0	0
1	1	1

Requirements

Version 1

See Also

Logical Operators | Not Operator | Operator Precedence | Operator Summary | Or Operator | Xor Operator

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Assignment Operator (=)

Assigns a value to a variable or property.

```
variable = value
```

Arguments

variable
    Any variable or any writable property.
value
    Any numeric or string literal, constant, or expression.

Remarks

The name on the left side of the equal sign can be a simple scalar variable or an element of an array. Properties on the left side of the equal sign can only be those properties that are writable at run time.

Requirements

Version 1

See Also

Comparison Operators | Operator Precedence | Operator Summary | Set Statement

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Concatenation Operator (&)

Forces string concatenation of two expressions.

```
result = expression1 & expression2
```

Arguments

result
    Any variable.
expression1
    Any expression.
expression2
    Any expression.

Remarks

Whenever an *expression* is not a string, it is converted to a **String** subtype. If both expressions are Null, *result* is also **Null**. However, if only one *expression* is **Null**, that expression is treated as a zero-length string ("") when concatenated with the other expression. Any expression that is Empty is also treated as a zero-length string.

Requirements

Version 1

See Also

Concatenation Operators | Operator Precedence | Operator Summary

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Division Operator (/)

Divides two numbers and returns a floating-point result.

```
result = number1/number2
```

Arguments

result
Any numeric variable.
number1
Any numeric expression.
number2
Any numeric expression.

Remarks

If one or both expressions are Null expressions, *result* is **Null**. Any expression that is Empty is treated as 0.

Requirements

Version 1

See Also

* Operator | \ Operator | Arithmetic Operators | Operator Precedence | Operator Summary

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Eqv Operator

Performs a logical equivalence on two expressions.

```
result = expression1 Eqv expression2
```

Arguments

result
    Any numeric variable.
expression1
    Any expression.
expression2
    Any expression.

Remarks

If either expression is Null, *result* is also **Null**. When neither expression is **Null**, *result* is determined according to the following table:

If expression1 is	And expression2 is	The *result* is
True	True	True
True	False	False
False	True	False
False	False	True

The **Eqv** operator performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following table:

If bit in *expression1* is	And bit in *expression2* is	The *result* is
0	0	1
0	1	0
1	0	0
1	1	1

Requirements

Version 1

See Also

Imp Operator | Logical Operators | Operator Precedence | Operator Summary

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Exponentiation Operator (^)

Raises a number to the power of an exponent.

```
result = number^exponent
```

Arguments

result
  Any numeric variable.
number
  Any numeric expression.
exponent
  Any numeric expression.

Remarks

Number can be negative only if *exponent* is an integer value. When more than one exponentiation is performed in a single expression, the ^ operator is evaluated as it is encountered from left to right.

If either *number* or *exponent* is a Null expression, *result* is also **Null**.

Requirements

Version 1

See Also

Arithmetic Operators | Operator Precedence | Operator Summary

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Imp Operator

Performs a logical implication on two expressions.

```
result = expression1 Imp expression2
```

Arguments

result
: Any numeric variable.

expression1
: Any expression.

expression2
: Any expression.

Remarks

The following table illustrates how *result* is determined:

If expression1 is	And expression2 is	Then *result* is
True	True	True
True	False	False
True	Null	Null
False	True	True
False	False	True
False	Null	True
Null	True	True
Null	False	Null
Null	Null	Null

The **Imp** operator performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following table:

If bit in *expression1* is	And bit in *expression2* is	Then *result* is
0	0	1
0	1	1
1	0	0
1	1	1

Requirements

Version 1

See Also

Eqv Operator | Logical Operators | Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Integer Division Operator (\)

Divides two numbers and returns an integer result.

```
result = number1\number2
```

Arguments

result
: Any numeric variable.

number1
: Any numeric expression.

number2
: Any numeric expression.

Remarks

Before division is performed, numeric expressions are rounded to **Byte**, **Integer**, or **Long** subtype expressions.

If any expression is Null, *result* is also **Null**. Any expression that is Empty is treated as 0.

Requirements

Version 1

See Also

* Operator | / Operator | Arithmetic Operators | Operator Precedence | Operator Summary

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Is Operator

Compares two object reference variables.

```
result = object1 Is object2
```

Arguments

result
    Any numeric variable.
object1
    Any object name.
object2
    Any object name.

Remarks

If *object1* and *object2* both refer to the same object, *result* is **True**; if they do not, *result* is **False**. Two variables can be made to refer to the same object in several ways.

In the following example, A has been set to refer to the same object as B:

```
Set A = B
```

The following example makes A and B refer to the same object as C:

```
Set A = C
Set B = C
```

Requirements

Version 1

See Also

Comparison Operators | Operator Precedence | Operator Summary

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Mod Operator

Divides two numbers and returns only the remainder.

```
result = number1 Mod number2
```

Arguments

result
    Any numeric variable.
number1
    Any numeric expression.
number2
    Any numeric expression.

Remarks

The modulus, or remainder, operator divides *number1* by *number2* (rounding floating-point numbers to integers) and returns only the remainder as *result*. For example, in the following expression, A (which is *result*) equals 5.

```
A = 19 Mod 6.7
```

If any expression is Null, *result* is also **Null**. Any expression that is Empty is treated as 0.

Requirements

Version 1

See Also

Arithmetic Operators | Operator Precedence | Operator Summary

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Multiplication Operator (*)

Multiplies two numbers.

```
result = number1*number2
```

Arguments

result
Any numeric variable.

number1
Any numeric expression.

number2
Any numeric expression.

Remarks

If one or both expressions are Null expressions, *result* is **Null**. If an expression is Empty, it is treated as if it were 0.

Requirements

Version 1

See Also

\ Operator | Arithmetic Operators | Operator Precedence | Operator Summary

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Not Operator

Performs logical negation on an expression.

```
result = Not expression
```

Arguments

result
  Any numeric variable.
expression
  Any expression.

Remarks

The following table illustrates how *result* is determined:

If expression is	Then *result* is
True	False
False	True
Null	Null

In addition, the **Not** operator inverts the bit values of any variable and sets the corresponding bit in *result* according to the following table:

Bit in expression	Bit in *result*
0	1
1	0

Requirements

Version 1

See Also

And Operator | Logical Operators | Operator Precedence | Operator Summary | Or Operator | Xor Operator

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Or Operator

Performs a logical disjunction on two expressions.

```
result = expression1 Or expression2
```

Arguments

result
  Any numeric variable.
expression1
  Any expression.
expression2
  Any expression.

Remarks

If either or both expressions evaluate to **True**, *result* is **True**. The following table illustrates how *result* is determined:

If expression1 is	And expression2 is	Then *result* is
True	True	True
True	False	True
True	Null	True
False	True	True
False	False	False
False	Null	Null
Null	True	True
Null	False	Null
Null	Null	Null

The **Or** operator also performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following table:

If bit in *expression1* is	And bit in *expression2* is	Then *result* is
0	0	0
0	1	1
1	0	1
1	1	1

Requirements

Version 1

See Also

And Operator | Logical Operators | Not Operator | Operator Precedence | Operator Summary | Xor Operator

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Subtraction Operator (-)

Finds the difference between two numbers or indicates the negative value of a numeric expression.

Syntax 1

```
result = number1-number2
```

Syntax 2

```
-number
```

Arguments

result
    Any numeric variable.
number
    Any numeric expression.
number1
    Any numeric expression.
number2
    Any numeric expression.

Remarks

In Syntax 1, the - operator is the arithmetic subtraction operator used to find the difference between two numbers. In Syntax 2, the - operator is used as the unary negation operator to indicate the negative value of an expression.

If one or both expressions are Null expressions, *result* is **Null**. If an expression is Empty, it is treated as if it were 0.

Requirements

Version 1

See Also

+ Operator | Arithmetic Operators | Operator Precedence | Operator Summary

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Xor Operator

Performs a logical exclusion on two expressions.

```
result = expression1 Xor expression2
```

Arguments

result
    Any numeric variable.
expression1

Any expression.

expression2

Any expression.

Remarks

If one, and only one, of the expressions evaluates to **True**, *result* is **True**. However, if either expression is Null, *result* is also **Null**. When neither expression is **Null**, *result* is determined according to the following table:

If expression1 is	And expression2 is	Then *result* is
True	True	False
True	False	True
False	True	True
False	False	False

The **Xor** operator also performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following table:

If bit in *expression1* is	And bit in *expression2* is	Then *result* is
0	0	0
0	1	1
1	0	1
1	1	0

Requirements

Version 1

See Also

And Operator | Logical Operators | Not Operator | Operator Precedence | Operator Summary | Or Operator

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Arithmetic Operators

^ Operator

* Operator

/ Operator

\ Operator

Mod Operator

+ Operator

- Operator

Concatenation Operators

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Comparison Operators

Used to compare expressions.

```
result = expression1 comparisonoperator expression2
result = object1 Is object2
```

Parts

result
Any numeric variable.
expression
Any expression.
comparisonoperator
Any comparison operator.
object
Any object name.

Remarks

The **Is** operator has specific comparison functionality that differs from the operators in the following table. The following table contains a list of the comparison operators and the conditions that determine whether *result* is **True**, **False**, or Null:

Operator	Description	True if	False if	Null if
<	Less than	*expression1* < *expression2*	*expression1* >= *expression2*	*expression1* or *expression2* = Null
<=	Less than or equal to	*expression1* <= *expression2*	*expression1* > *expression2*	*expression1* or *expression2* = Null
>	Greater than	*expression1* > *expression2*	*expression1* <= *expression2*	*expression1* or *expression2* = Null
>=	Greater than or equal to	*expression1* >= *expression2*	*expression1* < *expression2*	*expression1* or *expression2* = Null
=	Equal to	*expression1* = *expression2*	*expression1* <> *expression2*	*expression1* or *expression2* = Null
<>	Not equal to	*expression1* <> *expression2*	*expression1* = *expression2*	*expression1* or *expression2* = Null

When comparing two expressions, you may not be able to easily determine whether the expressions are being compared as numbers or as strings.

The following table shows how expressions are compared or what results from the comparison, depending on the underlying subtype:

If	Then
Both expressions are numeric	Perform a numeric comparison.
Both expressions are strings	Perform a string comparison.
One expression is numeric and the other is a string	The numeric expression is less than the string expression.
One expression is **Empty** and the other is numeric	Perform a numeric comparison, using 0 as the **Empty** expression.
One expression is **Empty** and the other is a string	Perform a string comparison, using a zero-length string ("") as the **Empty** expression.
Both expressions are **Empty**	The expressions are equal.

Requirements

Version 1

See Also

= Operator | Is Operator | Operator Precedence | Operator Summary

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Concatenation Operators

& Operator

+ Operator

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Logical Operators

And Operator

Not Operator

Or Operator

Xor Operator

© 2001 Microsoft Corporation. All rights reserved.

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Properties

In This Section

Description Property

FirstIndex Property

Global Property

HelpContext Property

HelpFile Property

IgnoreCase Property

Length Property

Number Property

Pattern Property

Source Property

Value Property

Related Sections

VBScript Langauge Reference

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Description Property

Returns or sets a descriptive string associated with an error.

```
object.Description [= stringexpression]
```

Arguments

object
Always the **Err** object.
stringexpression
A string expression containing a description of the error.

Remarks

The **Description** property consists of a short description of the error. Use this property to alert the user to an error that you can't or don't want to handle. When generating a user-defined error, assign a short description of your error to this property. If **Description** isn't filled in, and the value of **Number** corresponds to a VBScript run-time error, the descriptive string associated with the error is returned.

```
On Error Resume Next
Err.Raise 6   ' Raise an overflow error.
MsgBox ("Error # " & CStr(Err.Number) & " " & Err.Description)
Err.Clear   ' Clear the error.
```

Requirements

Version 1

See Also

Err Object | HelpContext Property | HelpFile Property | Number Property | Source Property

Applies To: Err Object

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

FirstIndex Property

Returns the position in a search string where a match occurs.

object.FirstIndex

The *object* argument is always a **Match** object.

Remarks

The **FirstIndex** property uses a zero-based offset from the beginning of the search string. In other words, the first character in the string is identified as character zero (0). The following code illustrates the use of the **FirstIndex** property.

```
Function RegExpTest(patrn, strng)
   Dim regEx, Match, Matches   ' Create variable.
   Set regEx = New RegExp   ' Create regular expression.
   regEx.Pattern = patrn   ' Set pattern.
   regEx.IgnoreCase = True   ' Set case insensitivity.
   regEx.Global = True   ' Set global applicability.
   Set Matches = regEx.Execute(strng)   ' Execute search.
   For Each Match in Matches   ' Iterate Matches collection.
      RetStr = RetStr & "Match " & I & " found at position "
      RetStr = RetStr & Match.FirstIndex & ". Match Value is "'
      RetStr = RetStr & Match.Value & "'." & vbCRLF
   Next
   RegExpTest = RetStr
End Function
MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
```

Requirements

Version 5

See Also

Length Property | Value Property

Applies To: [Match Object]

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Global Property

Sets or returns a **Boolean** value that indicates if a pattern should match all occurrences in an entire search string or just the first one.

object.**Global** [= **True** | **False**]

The *object* argument is always a **RegExp** object. The value of the **Global** property is **True** if the search applies to the entire string, **False** if it does not. Default is **False**.

Remarks

The following code illustrates the use of the **Global** property (change the value assigned to **Global** property to see its effect):

```
Function RegExpTest(patrn, strng)
   Dim regEx, Match, Matches   ' Create variable.
   Set regEx = New RegExp   ' Create a regular expression.
   regEx.Pattern = patrn   ' Set pattern.
   regEx.IgnoreCase = True   ' Set case insensitivity.
   regEx.Global = True   ' Set global applicability.
   Set Matches = regEx.Execute(strng)   ' Execute search.
   For Each Match in Matches   ' Iterate Matches collection.
      RetStr = RetStr & "Match found at position "
      RetStr = RetStr & Match.FirstIndex & ". Match Value is '"
      RetStr = RetStr & Match.Value & "'." & vbCRLF
   Next
   RegExpTest = RetStr
End Function
```

```
MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
```

Requirements

Version 5

See Also

IgnoreCase Property | Pattern Property

Applies To: RegExp Object

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

HelpContext Property

Sets or returns a context ID for a topic in a Help File.

object.**HelpContext** [= *contextID*]

Arguments

object
: Required. Always the **Err** object.

contextID
: Optional. A valid identifier for a Help topic within the Help file.

Remarks

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

The following example illustrates use of the **HelpContext** property:

```
On Error Resume Next
Dim Msg
Err.Clear
Err.Raise 6   ' Generate "Overflow" error.
Err.Helpfile = "yourHelp.hlp"
Err.HelpContext = yourContextID
If Err.Number <> 0 Then
   Msg = "Press F1 or Help to see " & Err.Helpfile & " topic for" & _
   " the following HelpContext: " & Err.HelpContext
   MsgBox Msg, , "error: " & Err.Description, Err.Helpfile, Err.HelpContext
End If
```

Requirements

Version 2

See Also

Description Property | HelpFile Property | Number Property | Source Property

Applies To: Err Object

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

HelpFile Property

Sets or returns a fully qualified path to a Help File.

```
object.HelpFile [= contextID]
```

Arguments

object
Required. Always the **Err** object.

contextID
Optional. Fully qualified path to the Help file.

Remarks

If a Help file is specified in **HelpFile**, it is automatically called when the user clicks the Help button (or presses the F1 key) in the error message dialog box. If the **HelpContext** property contains a valid context ID for the specified file, that topic is automatically displayed. If no **HelpFile** is specified, the VBScript Help file is displayed.

```
On Error Resume Next
Dim Msg
Err.Clear
Err.Raise 6   ' Generate "Overflow" error.
Err.Helpfile = "yourHelp.hlp"
Err.HelpContext = yourContextID
If Err.Number <> 0 Then
   Msg = "Press F1 or Help to see " & Err.Helpfile & " topic for" & _
   " the following HelpContext: " & Err.HelpContext
   MsgBox Msg, , "error: " & Err.Description, Err.Helpfile, Err.HelpContext
End If
```

Requirements

Version 2

See Also

Description Property | HelpContext Property | Number Property | Source Property

Applies To: Err Object

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

IgnoreCase Property

Sets or returns a **Boolean** value that indicates if a pattern search is case-sensitive or not.

```
object.IgnoreCase [= True | False ]
```

The *object* argument is always a **RegExp** object. The value of the **IgnoreCase** property is **False** if the search is case-sensitive, **True** if it is not. Default is **False**.

Remarks

The following code illustrates the use of the **IgnoreCase** property (change the value assigned to **IgnoreCase** property to see its effect):

```
Function RegExpTest(patrn, strng)
   Dim regEx, Match, Matches   ' Create variable.
   Set regEx = New RegExp   ' Create a regular expression.
   regEx.Pattern = patrn   ' Set pattern.
   regEx.IgnoreCase = True   ' Set case insensitivity.
   regEx.Global = True   ' Set global applicability.
   Set Matches = regEx.Execute(strng)   ' Execute search.
   For Each Match in Matches   ' Iterate Matches collection.
      RetStr = RetStr & "Match found at position "
      RetStr = RetStr & Match.FirstIndex & ". Match Value is '"
      RetStr = RetStr & Match.Value & "'." & vbCRLF
```

```
   Next
   RegExpTest = RetStr
End Function
MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
```

Requirements

Version 5

See Also

Global Property | Pattern Property

Applies To: RegExp Object

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Length Property

Returns the length of a match found in a search string.

`object.Length`

The *object* argument is always a **Match** object.

Remarks

The following code illustrates the use of the **Length** property:

```
Function RegExpTest(patrn, strng)
   Dim regEx, Match, Matches   ' Create variable.
   Set regEx = New RegExp   ' Create regular expression.
   regEx.Pattern = patrn   ' Set pattern.
   regEx.IgnoreCase = True   ' Set case insensitivity.
   regEx.Global = True   ' Set global applicability.
   Set Matches = regEx.Execute(strng)   ' Execute search.
   For Each Match in Matches   ' Iterate Matches collection.
      RetStr = RetStr & "Match " & I & " found at position "
      RetStr = RetStr & Match.FirstIndex & ". Match Length is "
      RetStr = RetStr & Match.Length
      RetStr = RetStr & " characters." & vbCRLF
   Next
   RegExpTest = RetStr
End Function
MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
```

Requirements

Version 5

See Also

FirstIndex Property | Value Property

Applies To: Match Object

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Number Property

Returns or sets a numeric value specifying an error. **Number** is the **Err** object's default property.

```
object.Number [= errornumber]
```

Arguments

object
Always the **Err** object.
errornumber
An integer representing a VBScript error number or an SCODE error value.

Remarks

When returning a user-defined error from an Automation object, set **Err.Number** by adding the number you selected as an error code to the constant **vbObjectError**.

The following code illustrates the use of the **Number** property.

```
On Error Resume Next
Err.Raise vbObjectError + 1, "SomeObject"    ' Raise Object Error #1.
MsgBox ("Error # " & CStr(Err.Number) & " " & Err.Description)
Err.Clear   ' Clear the error.
```

Requirements

Version 1

See Also

Description Property | HelpContext Property | HelpFile Property | Err Object | Source Property

Applies To: Err Object

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Pattern Property

Sets or returns the regular expression pattern being searched for.

```
object.Pattern [= "searchstring"]
```

Arguments

object
Required. Always a **RegExp** object variable.

searchstring
Optional. Regular string expression being searched for. May include any of the regular expression characters defined in the table in the **Settings** section.

Settings

Special characters and sequences are used in writing patterns for regular expressions. The following table describes and gives an example of the characters and sequences that can be used.

<table>
<tr><th>Character</th><th>Description</th></tr>
<tr><td>\</td><td>Marks the next character as either a special character or a literal. For example, "n" matches the character "n". "\n" matches a newline character. The sequence "\\" matches "\" and "\(" matches "(".</td></tr>
<tr><td>^</td><td>Matches the beginning of input.</td></tr>
<tr><td>$</td><td>Matches the end of input.</td></tr>
<tr><td>*</td><td>Matches the preceding character zero or more times. For example, "zo*" matches either "z" or "zoo".</td></tr>
<tr><td>+</td><td>Matches the preceding character one or more times. For example, "zo+" matches "zoo" but not "z".</td></tr>
<tr><td>?</td><td>Matches the preceding character zero or one time. For example, "a?ve?" matches the "ve" in "never".</td></tr>
<tr><td>.</td><td>Matches any single character except a newline character.</td></tr>
<tr><td>(pattern)</td><td>Matches pattern and remembers the match. The matched substring can be retrieved from the resulting Matches collection, using Item [0]...[n]. To match parentheses characters ( ), use "\(" or "\)".</td></tr>
<tr><td>x|y</td><td>Matches either x or y. For example, "z|wood" matches "z" or "wood". "(z|w)oo" matches "zoo" or "wood".</td></tr>
</table>

{*n*}	*n* is a nonnegative integer. Matches exactly *n* times. For example, "o{2}" does not match the "o" in "Bob," but matches the first two o's in "foooood".
{*n*,}	*n* is a nonnegative integer. Matches at least *n* times. For example, "o{2,}" does not match the "o" in "Bob" and matches all the o's in "foooood." "o{1,}" is equivalent to "o+". "o{0,}" is equivalent to "o*".
{*n*,*m*}	*m* and *n* are nonnegative integers. Matches at least *n* and at most *m* times. For example, "o{1,3}" matches the first three o's in "fooooood." "o{0,1}" is equivalent to "o?".
[*xyz*]	A character set. Matches any one of the enclosed characters. For example, "[abc]" matches the "a" in "plain".
[^*xyz*]	A negative character set. Matches any character not enclosed. For example, "[^abc]" matches the "p" in "plain".
[*a-z*]	A range of characters. Matches any character in the specified range. For example, "[a-z]" matches any lowercase alphabetic character in the range "a" through "z".
[^*m-z*]	A negative range characters. Matches any character not in the specified range. For example, "[m-z]" matches any character not in the range "m" through "z".
\b	Matches a word boundary, that is, the position between a word and a space. For example, "er\b" matches the "er" in "never" but not the "er" in "verb".
\B	Matches a non-word boundary. "ea*r\B" matches the "ear" in "never early".
\d	Matches a digit character. Equivalent to [0-9].
\D	Matches a non-digit character. Equivalent to [^0-9].
\f	Matches a form-feed character.
\n	Matches a newline character.
\r	Matches a carriage return character.
\s	Matches any white space including space, tab, form-feed, etc. Equivalent to "[\f\n\r\t\v]".
\S	Matches any nonwhite space character. Equivalent to "[^ \f\n\r\t\v]".
\t	Matches a tab character.
\v	Matches a vertical tab character.
\w	Matches any word character including underscore. Equivalent to "[A-Za-z0-9_]".
\W	Matches any non-word character. Equivalent to "[^A-Za-z0-9_]".
\num	Matches *num*, where *num* is a positive integer. A reference back to remembered matches. For example, "(.)\1" matches two consecutive identical characters.
n	Matches *n*, where *n* is an octal escape value. Octal escape values must be 1, 2, or 3 digits long. For example, "\11" and "\011" both match a tab character. "\0011" is the equivalent of "\001" & "1". Octal escape values must not exceed 256. If they do, only the first two digits comprise the expression. Allows ASCII codes to be used in regular expressions.
\x*n*	Matches *n*, where *n* is a hexadecimal escape value. Hexadecimal escape values must be exactly two digits long. For example, "\x41" matches "A". "\x041" is equivalent to "\x04" & "1". Allows ASCII codes to be used in regular expressions.

Remarks

The following code illustrates the use of the **Pattern** property.

```
Function RegExpTest(patrn, strng)
   Dim regEx, Match, Matches   ' Create variable.
   Set regEx = New RegExp   ' Create a regular expression.
   regEx.Pattern = patrn   ' Set pattern.
   regEx.IgnoreCase = True   ' Set case insensitivity.
   regEx.Global = True   ' Set global applicability.
   Set Matches = regEx.Execute(strng)   ' Execute search.
   For Each Match in Matches   ' Iterate Matches collection.
      RetStr = RetStr & "Match found at position "
      RetStr = RetStr & Match.FirstIndex & ". Match Value is '"
      RetStr = RetStr & Match.Value & "'." & vbCRLF
   Next
   RegExpTest = RetStr
End Function
MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
```

Requirements

Version 2

See Also

Global Property | IgnoreCase Property

Applies To: RegExp Object

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Source Property

Returns or sets the name of the object or application that originally generated the error.

```
object.Source [= stringexpression]
```

Arguments

object
: Always the **Err** object.

stringexpression
: A string expression representing the application that generated the error.

Remarks

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

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

When generating an error from code, **Source** is your application's programmatic ID.

The following code illustrates use of the **Source** property.

```
On Error Resume Next
Err.Raise 6   ' Raise an overflow error.
MsgBox ("Error # " & CStr(Err.Number) & " " & Err.Description & Err.Source)
Err.Clear   ' Clear the error.
```

Requirements

Version 1

See Also

Description Property | Err Object | HelpContext Property | HelpFile Property | Number Property | On Error Statement

Applies To: Err Object

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Value Property

Returns the value or text of a match found in a search string.

object.**Value**

The *object* argument is always a **Match** object.

Remarks

The following code illustrates the use of the **Value** property.

```
Function RegExpTest(patrn, strng)
   Dim regEx, Match, Matches   ' Create variable.
   Set regEx = New RegExp   ' Create regular expression.
   regEx.Pattern = patrn   ' Set pattern.
   regEx.IgnoreCase = True   ' Set case insensitivity.
   regEx.Global = True   ' Set global applicability.
   Set Matches = regEx.Execute(strng)   ' Execute search.
   For Each Match in Matches   ' Iterate Matches collection.
```

```
      RetStr = RetStr & "Match " & I & " found at position "
      RetStr = RetStr & Match.FirstIndex & ". Match Value is "'
      RetStr = RetStr & Match.Value & "'." & vbCRLF
   Next
   RegExpTest = RetStr
End Function
MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
```

Requirements

Version 1

See Also

FirstIndex Property | Length Property

Applies To: Match Object

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Statements

In This Section

Call Statement

Class Statement

Const Statement

Related Sections

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Call Statement

Transfers control to a **Sub** or **Function** procedure.

```
[Call] name [argumentlist]
```

Arguments

Call
: Optional keyword. If specified, you must enclose *argumentlist* in parentheses. For example:

    Call MyProc(0)

name
: Required. Name of the procedure to call.

argumentlist
: Optional. Comma-delimited list of variables, arrays, or expressions to pass to the procedure.

Remarks

You are not required to use the **Call** keyword when calling a procedure. However, if you use the **Call** keyword to call a procedure that requires arguments, *argumentlist* must be enclosed in parentheses. If you omit the **Call** keyword, you also must omit the parentheses around *argumentlist*. If you use either **Call** syntax to call any intrinsic or user-defined function, the function's return value is discarded.

```
Call MyFunction("Hello World")
Function MyFunction(text)
   MsgBox text
End Function
```

Requirements

Version 1

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Class Statement

Declares the name of a class, as well as a definition of the variables, properties, and methods that comprise the class.

```
Class name
   statements
End Class
```

Arguments

name

Required. Name of the **Class**; follows standard variable naming conventions.

statements

Required. One or more statements that define the variables, properties, and methods of the **Class**.

Remarks

Within a **Class** block, members are declared as either **Private** or **Public** using the appropriate declaration statements. Anything declared as **Private** is visible only within the **Class** block. Anything declared as Public is visible within the **Class** block, as well as by code outside the **Class** block. Anything not explicitly declared as either **Private** or **Public** is **Public** by default. Procedures (either **Sub** or **Function**) declared **Public** within the class block become methods of the class. **Public** variables serve as properties of the class, as do properties explicitly declared using **Property Get**, **Property Let**, and **Property Set**. Default properties and methods for the class are specified in their declarations using the **Default** keyword. See the individual declaration statement topics for information on how this keyword is used.

Requirements

Version 5

See Also

Dim Statement | Function Statement | Private Statement | Property Get Statement | Property Let Statement | Property Set Statement | Public Statement | Set Statement | Sub Statement

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Const Statement

Declares constants for use in place of literal values.

```
[Public | Private] Const constname = expression
```

Arguments

Public
: Optional. Keyword used at script level to declare constants that are available to all procedures in all scripts. Not allowed in procedures.

Private
: Optional. Keyword used at script level to declare constants that are available only within the script where the declaration is made. Not allowed in procedures.

constname
: Required. Name of the constant; follows standard variable naming conventions.

expression
: Required. Literal or other constant, or any combination that includes all arithmetic or logical operators except **Is**.

Remarks

Constants are public by default. Within procedures, constants are always private; their visibility can't be changed. Within a script, the default visibility of a script-level constant can be changed using the **Private** keyword.

To combine several constant declarations on the same line, separate each constant assignment with a comma. When constant declarations are combined in this way, the **Public** or **Private** keyword, if used, applies to all of them.

You can't use variables, user-defined functions, or intrinsic VBScript functions (such as **Chr**) in constant declarations. By definition, they can't be constants. You also can't create a constant from any expression that involves an operator, that is, only simple constants are allowed. Constants declared in a **Sub** or **Function** procedure are local to that procedure. A constant declared outside a procedure is defined throughout the script in which it is declared. You can use constants anywhere you can use an expression. The following code illustrates the use of the **Const** statement:

```
Const MyVar = 459   ' Constants are Public by default.
Private Const MyString = "HELP"   ' Declare Private constant.
```

```
Const MyStr = "Hello", MyNumber = 3.4567   ' Declare multiple constants on same line.
```

> **Note** Constants can make your scripts self-documenting and easy to modify. Unlike variables, constants cannot be inadvertently changed while your script is running.

Requirements

Version 5

See Also

Dim Statement | Function Statement | Private Statement | Public Statement | Sub Statement

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Dim Statement

Declares variables and allocates storage space.

```
Dim varname[([subscripts])][, varname[([subscripts])]] . . .
```

Arguments

varname
: Name of the variable; follows standard variable naming conventions.

subscripts
: Dimensions of an array variable; up to 60 multiple dimensions may be declared. The *subscripts* argument uses the following syntax:

  upperbound [**,**upperbound] **. . .**

The lower bound of an array is always zero.

Remarks

Variables declared with **Dim** at the script level are available to all procedures within the script. At the procedure level, variables are available only within the procedure.

You can also use the **Dim** statement with empty parentheses to declare a dynamic array. After declaring a dynamic array, use the **ReDim** statement within a procedure to define the number of dimensions and elements in the array. If you try to redeclare a dimension for an array variable whose size was explicitly specified in a **Dim** statement, an error occurs.

Note When you use the **Dim** statement in a procedure, you generally put the **Dim** statement at the beginning of the procedure.

The following examples illustrate the use of the **Dim** statement:

```
Dim Names(9)       ' Declare an array with 10 elements.
Dim Names()        ' Declare a dynamic array.
Dim MyVar, MyNum   ' Declare two variables.
```

Requirements

Version 1

See Also

Private Statement | Public Statement | ReDim Statement | Set Statement

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Do...Loop Statement

Repeats a block of statements while a condition is **True** or until a condition becomes **True**.

```
Do [{While | Until} condition]
   [statements]
   [Exit Do]
   [statements]
Loop
```

Or, you can use this syntax:

```
Do
   [statements]
   [Exit Do]
   [statements]
Loop [{While | Until} condition]
```

Arguments

condition
    Numeric or string expression that is **True** or **False**. If *condition* is **Null**, *condition* is treated as **False**.
statements
    One or more statements that are repeated while or until *condition* is **True**.

Remarks

The **Exit Do** can only be used within a **Do...Loop** control structure to provide an alternate way to exit a **Do...Loop**. Any number of **Exit Do** statements may be placed anywhere in the **Do...Loop**. Often used with the evaluation of some condition (for example, **If...Then**), **Exit Do** transfers control to the statement immediately following the **Loop**.

When used within nested **Do...Loop** statements, **Exit Do** transfers control to the loop that is nested one level above the loop where it occurs.

The following examples illustrate use of the **Do...Loop** statement:

```
Do Until DefResp = vbNo
   MyNum = Int (6 * Rnd + 1)   ' Generate a random integer between 1 and 6.
   DefResp = MsgBox (MyNum & " Do you want another number?", vbYesNo)
```

```
Loop

Dim Check, Counter
Check = True: Counter = 0   ' Initialize variables.
Do                           ' Outer loop.
   Do While Counter < 20     ' Inner loop.
      Counter = Counter + 1  ' Increment Counter.
      If Counter = 10 Then   ' If condition is True...
         Check = False       ' set value of flag to False.
         Exit Do             ' Exit inner loop.
      End If
   Loop
Loop Until Check = False     ' Exit outer loop immediately.
```

Requirements

Version 1

See Also

Exit Statement | For...Next Statement | While...Wend Statement

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Erase Statement

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

Erase *array*

The *array* argument is the name of the array variable to be erased.

Remarks

It is important to know whether an array is fixed-size (ordinary) or dynamic because **Erase** behaves differently depending on the type of array. **Erase** recovers no memory for fixed-size arrays. **Erase** sets the elements of a fixed array as follows:

Type of array	Effect of Erase on fixed-array elements
Fixed numeric array	Sets each element to zero.
Fixed string array	Sets each element to zero-length ("").
Array of objects	Sets each element to the special value Nothing.

Erase frees the memory used by dynamic arrays. Before your program can refer to the dynamic array again, it must redeclare the array variable's dimensions using a **ReDim** statement.

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.
```

Requirements

Version 1

See Also

Dim Statement | Nothing | ReDim Statement

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Execute Statement

Executes one or more specified statements.

`Execute` *`statement`*

The required *statement* argument is a string expression containing one or more statements for execution. Include multiple statements in the *statement* argument, using colons or embedded line breaks to separate them.

Remarks

In VBScript, *x* = *y* can be interpreted two ways. The first is as an assignment statement, where the value of *y* is assigned to *x*. The second interpretation is as an expression that tests if *x* and *y* have the same value. If they do, *result* is **True**; if they are not, *result* is **False**. The **Execute** statement always uses the first interpretation, whereas the **Eval** method always uses the second.

> **Note** In Microsoft® JScript™, no confusion exists between assignment and comparison, because the assignment operator (=) is different from the comparison operator(==).

The context in which the **Execute** statement is invoked determines what objects and variables are available to the code being run. In-scope objects and variables are available to code running in an **Execute** statement. However, it is important to understand that if you execute code that creates a procedure, that procedure does not inherit the scope of the procedure in which it occurred.

Like any procedure, the new procedure's scope is global, and it inherits everything in the global scope. Unlike any other procedure, its context is not global scope, so it can only be executed in the context of the procedure where the **Execute** statement occurred. However, if the same **Execute** statement is invoked outside of a procedure (i.e., in global scope), not only does it inherit everything in global scope, but it can also be called from anywhere, since its context is global. The following example illustrates this behavior:

```
Dim X   ' Declare X in global scope.
X = "Global"   ' Assign global X a value.
Sub Proc1   ' Declare procedure.
   Dim X   ' Declare X in local scope.
   X = "Local"   ' Assign local X a value.
            ' The Execute statement here creates a
            ' procedure that, when invoked, prints X.
            ' It print the global X because Proc2
            ' inherits everything in global scope.
   Execute "Sub Proc2: Print X: End Sub"
   Print Eval("X")   ' Print local X.
```

```
   Proc2   ' Invoke Proc2 in Proc1's scope.
End Sub
Proc2   ' This line causes an error since
        ' Proc2 is unavailable outside Proc1.
Proc1   ' Invoke Proc1.
   Execute "Sub Proc2: Print X: End Sub"
Proc2   ' This invocation succeeds because Proc2
        ' is now available globally.
```

The following example shows how the **Execute** statement can be rewritten so you don't have to enclose the entire procedure in the quotation marks:

```
S = "Sub Proc2" & vbCrLf
S = S & "   Print X" & vbCrLf
S = S & "End Sub"
Execute S
```

Requirements

Version 1

See Also

Eval Function | ExecuteGlobal Statement

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

ExecuteGlobal Statement

Executes one or more specified statements in the global namespace of a script.

```
ExecuteGlobal statement
```

The required *statement* argument is a string expression containing one or more statements for execution. Include multiple statements in the *statement* argument, using colons or embedded line breaks to separate them.

Remarks

In VBScript, *x* = *y* can be interpreted two ways. The first is as an assignment statement, where the value of *y* is assigned to *x*. The second interpretation is as an expression that tests if *x* and *y* have the same value. If they do, *result* is **True**; if they are not, *result* is **False**. The **ExecuteGlobal** statement always uses the first interpretation, whereas the **Eval** method always uses the second.

> **Note** In Microsoft® JScript™, no confusion exists between assignment and comparison, because the assignment operator (=) is different from the comparison operator.

All statements used with **ExecuteGlobal** are executed in the script's global namespace. This allows code to be added to the program so that any procedure can access it. For example, a VBScript **Class** statement can be executed at run time and functions can subsequently create new instances of the class.

Adding procedures and classes at runtime can be useful, but also introduces the possibility of overwriting existing global variable and functions at runtime. Because this can cause significant programming problems, care should be exercised when using the **ExecuteGlobal** statement. If you don't need access to a variable or function outside of a procedure, use the **Execute** statement that will only affect the namespace of the calling function.

The following example illustrates the use of the **ExecuteGlobal** statement:

```
Dim X          ' Declare X in global scope.
X = "Global"       ' Assign global X a value.
Sub Proc1   ' Declare procedure.
  Dim X      ' Declare X in local scope.
  X = "Local"   ' Assign local X a value.
         ' The Execute statement here creates a
         ' procedure that, when invoked, prints X.
         ' It print the global X because Proc2
         ' inherits everything in global scope.
  ExecuteGlobal "Sub Proc2: Print X: End Sub"
  Print Eval("X")   ' Print local X.
  Proc2      ' Invoke Proc2 in Global scope resulting
         ' in "Global" being printed.
End Sub
```

```
Proc2          ' This line causes an error since
         ' Proc2 is unavailable outside Proc1.
Proc1          ' Invoke Proc1.
  Execute "Sub Proc2: Print X: End Sub"
Proc2          ' This invocation succeeds because Proc2
         ' is now available globally.
```

The following example shows how the **ExecuteGlobal** statement can be rewritten so you don't have to enclose the entire procedure in the quotation marks:

```
S = "Sub Proc2" & vbCrLf
S = S & "  Print X" & vbCrLf
S = S & "End Sub"
ExecuteGlobal S
```

Requirements

Version 1

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Exit Statement

Exits a block of **Do...Loop**, **For...Next**, **Function**, or **Sub** code.

```
Exit Do
Exit For
Exit Function
Exit Property
Exit Sub
```

The **Exit** statement syntax has these forms:

Statement	Description
Exit Do	Provides a way to exit a **Do...Loop** statement. It can be used only inside a **Do...Loop** statement. **Exit Do** transfers control to the statement following the **Loop** statement. When used within nested **Do...Loop** statements, **Exit Do** transfers control to the loop that is one nested level above the loop where it occurs.
Exit For	Provides a way to exit a **For** loop. It can be used only in a **For...Next** or **For Each...Next** loop. **Exit For** transfers control to the statement following the **Next** statement. When used within nested **For** loops, **Exit For** transfers control to the loop that is one nested level above the loop where it occurs.
Exit Function	Immediately exits the **Function** procedure in which it appears. Execution continues with the statement following the statement that called the **Function**.
Exit Property	Immediately exits the **Property** procedure in which it appears. Execution continues with the statement following the statement that called the **Property** procedure.
Exit Sub	Immediately exits the **Sub** procedure in which it appears. Execution continues with the statement following the statement that called the **Sub**.

The following example illustrates the use of the **Exit** statement:

```
Sub RandomLoop
   Dim I, MyNum
   Do   ' Set up infinite loop.
      For I = 1 To 1000   ' Loop 1000 times.
         MyNum = Int(Rnd * 100)   ' Generate random numbers.
         Select Case MyNum   ' Evaluate random number.
            Case 17: MsgBox "Case 17"
               Exit For   ' If 17, exit For...Next.
            Case 29: MsgBox "Case 29"
               Exit Do   ' If 29, exit Do...Loop.
            Case 54: MsgBox "Case 54"
               Exit Sub   ' If 54, exit Sub procedure.
            End Select
      Next
   Loop
End Sub
```

Requirements

Version 1

See Also

Do...Loop Statement | For Each...Next Statement | For...Next Statement | Function Statement | Sub Statement

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

For Each...Next Statement

Repeats a group of statements for each element in an array or collection.

```
For Each element In group
   [statements]
   [Exit For]
   [statements]
Next [element]
```

Arguments

element
: Variable used to iterate through the elements of the collection or array. For collections, *element* can only be a **Variant** variable, a generic **Object** variable, or any specific Automation object variable. For arrays, *element* can only be a **Variant** variable.

group
: Name of an object collection or array.

statements
: One or more statements that are executed on each item in *group*.

Remarks

The **For Each** block is entered if there is at least one element in *group*. Once the loop has been entered, all the statements in the loop are executed for the first element in *group*. As long as there are more elements in *group*, the statements in the loop continue to execute for each element. When there are no more elements in *group*, the loop is exited and execution continues with the statement following the **Next** statement.

The **Exit For** can only be used within a **For Each...Next** or **For...Next** control structure to provide an alternate way to exit. Any number of **Exit For** statements may be placed anywhere in the loop. The **Exit For** is often used with the evaluation of some condition (for example, **If...Then**), and transfers control to the statement immediately following **Next**.

You can nest **For Each...Next** loops by placing one **For Each...Next** loop within another. However, each loop *element* must be unique.

> **Note** If you omit *element* in a **Next** statement, execution continues as if you had included it. If a **Next** statement is encountered before it's corresponding **For** statement, an error occurs.

The following example illustrates use of the **For Each...Next** statement:

```
Function ShowFolderList(folderspec)
   Dim fso, f, f1, fc, s
   Set fso = CreateObject("Scripting.FileSystemObject")
   Set f = fso.GetFolder(folderspec)
   Set fc = f.Files
   For Each f1 in fc
      s = s & f1.name
      s = s & "<BR>"
   Next
   ShowFolderList = s
End Function
```

Requirements

Version 2

See Also

Do...Loop Statement | Exit Statement | For...Next Statement | While...Wend Statement

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

For...Next Statement

Repeats a group of statements a specified number of times.

```
For counter = start To end [Step step]
    [statements]
    [Exit For]
    [statements]
Next
```

Arguments

counter
: Numeric variable used as a loop counter. The variable can't be an array element or an element of a user-defined type.

start
: Initial value of *counter*.

end
: Final value of *counter*.

step
: Amount *counter* is changed each time through the loop. If not specified, *step* defaults to one.

statements
: One or more statements between **For** and **Next** that are executed the specified number of times.

Remarks

The *step* argument can be either positive or negative. The value of the *step* argument determines loop processing as follows:

Value	Loop executes if
Positive or 0	counter <= end
Negative	counter >= end

Once the loop starts and all statements in the loop have executed, *step* is added to *counter*. At this point, either the statements in the loop execute again (based on the same test that caused the loop to execute initially), or the loop is exited and execution continues with the statement following the **Next** statement.

> **Note** Changing the value of *counter* while inside a loop can make it more difficult to read and debug your code.

Exit For can only be used within a **For Each...Next** or **For...Next** control structure to provide an alternate way to exit. Any number of **Exit For** statements may be placed anywhere in the loop. **Exit For** is often used with the evaluation of some condition (for example, **If...Then**), and transfers control to the statement immediately following **Next**.

You can nest **For...Next** loops by placing one **For...Next** loop within another. Give each loop a unique variable name as its *counter*. The following construction is correct:

```
For I = 1 To 10
      For J = 1 To 10
            For K = 1 To 10
            . . .
            Next
      Next
Next
```

Requirements

Version 1

See Also

Do...Loop Statement | Exit Statement | For Each...Next Statement | While...Wend Statement

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Function Statement

Declares the name, arguments, and code that form the body of a **Function** procedure.

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

Arguments

Public
: Indicates that the **Function** procedure is accessible to all other procedures in all scripts.

Default
: Used only with the **Public** keyword in a **Class** block to indicate that the **Function** procedure is the default method for the class. An error occurs if more than one **Default** procedure is specified in a class.

Private
: Indicates that the **Function** procedure is accessible only to other procedures in the script where it is declared or if the function is a member of a class, and that the **Function** procedure is accessible only to other procedures in that class.

name
: Name of the **Function**; follows standard variable naming conventions.

arglist
: List of variables representing arguments that are passed to the **Function** procedure when it is called. Commas separate multiple variables.

statements
: Any group of statements to be executed within the body of the **Function** procedure.

expression
: Return value of the **Function**.

The *arglist* argument has the following syntax and parts:

[**ByVal** | **ByRef**] *varname*[**( )**]

Arguments

ByVal
Indicates that the argument is passed by value.

ByRef
Indicates that the argument is passed by reference.

varname
Name of the variable representing the argument; follows standard variable naming conventions.

Remarks

If not explicitly specified using either **Public** or **Private**, **Function** procedures are public by default, that is, they are visible to all other procedures in your script. The value of local variables in a **Function** is not preserved between calls to the procedure.

You cannot define a **Function** procedure inside any other procedure (e.g. **Sub** or **Property Get**).

The **Exit Function** statement causes an immediate exit from a **Function** procedure. Program execution continues with the statement that follows the statement that called the **Function** procedure. Any number of **Exit Function** statements can appear anywhere in a **Function** procedure.

Like a **Sub** procedure, a **Function** procedure is a separate procedure that can take arguments, perform a series of statements, and change the values of its arguments. However, unlike a **Sub** procedure, you can use a **Function** procedure on the right side of an expression in the same way you use any intrinsic function, such as **Sqr**, **Cos**, or **Chr**, when you want to use the value returned by the function.

You call a **Function** procedure using the function name, followed by the argument list in parentheses, in an expression. See the **Call** statement for specific information on how to call **Function** procedures.

> **Caution** **Function** procedures can be recursive, that is, they can call themselves to perform a given task. However, recursion can lead to stack overflow.

To return a value from a function, assign the value to the function name. Any number of such assignments can appear anywhere within the procedure. If no value is assigned to *name*, the procedure returns a default value: a numeric function returns 0 and a string function returns a zero-length string (""). A function that returns an object reference returns **Nothing** if no object reference is assigned to *name* (using **Set**) within the **Function**.

The following example shows how to assign a return value to a function named BinarySearch. In this case, **False** is assigned to the name to indicate that some value was not found.

```
Function BinarySearch(. . .)
```

```
      . . .
      ' Value not found. Return a value of False.
      If lower > upper Then
            BinarySearch = False
            Exit Function
      End If
      . . .
End Function
```

Variables used in **Function** procedures fall into two categories: those that are explicitly declared within the procedure and those that are not. Variables that are explicitly declared in a procedure (using **Dim** or the equivalent) are always local to the procedure. Variables that are used but not explicitly declared in a procedure are also local unless they are explicitly declared at some higher level outside the procedure.

> **Caution** A procedure can use a variable that is not explicitly declared in the procedure, but a naming conflict can occur if anything you have defined at the script level has the same name. If your procedure refers to an undeclared variable that has the same name as another procedure, constant, or variable, it is assumed that your procedure is referring to that script-level name. To avoid this kind of conflict, use an **Option Explicit** statement to force explicit declaration of variables.

> **Caution** VBScript may rearrange arithmetic expressions to increase internal efficiency. Avoid using a **Function** procedure in an arithmetic expression when the function changes the value of variables in the same expression.

Requirements

Version 1

See Also

Call Statement | Dim Statement | Exit Statement | Nothing | Set Statement | Sub Statement

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

If...Then...Else Statement

Conditionally executes a group of statements, depending on the value of an expression.

```
If condition Then statements [Else elsestatements ]
```

Or, you can use the block form syntax:

```
If condition Then
   [statements]
[ElseIf condition-n Then
   [elseifstatements]] . . .
[Else
   [elsestatements]]
End If
```

Arguments

condition
: One or more of the following two types of expressions:

  A numeric or string expression that evaluates to **True** or **False**. If *condition* is Null, *condition* is treated as **False**.

  An expression of the form **TypeOf** *objectname* **Is** *objecttype*. The *objectname* is any object reference and *objecttype* is any valid object type. The expression is **True** if *objectname* is of the object type specified by *objecttype*; otherwise it is **False**.

statements
: One or more statements separated by colons; executed if *condition* is **True**.

condition-n
: Same as *condition*.

elseifstatements
: One or more statements executed if the associated *condition-n* is **True**.

elsestatements
: One or more statements executed if no previous *condition* or *condition-n* expression is **True**.

Remarks

You can use the single-line form (first syntax) for short, simple tests. However, the block form (second syntax) provides more structure and flexibility than the single-line form and is usually easier to read, maintain, and debug.

> **Note** With the single-line syntax, it is possible to have multiple statements executed as the result of an **If...Then** decision, but they must all be on the same line and separated by colons, as in the following statement:

```
If A > 10 Then A = A + 1 : B = B + A : C = C + B
```

When executing a block **If** (second syntax), *condition* is tested. If *condition* is **True**, the statements following **Then** are executed. If *condition* is **False**, each **ElseIf** (if any) is evaluated in turn. When a **True** condition is found, the statements following the associated **Then** are executed. If none of the **ElseIf** statements are **True** (or there are no **ElseIf** clauses), the statements following **Else** are executed. After executing the statements following **Then** or **Else**, execution continues with the statement following **End If**.

The **Else** and **ElseIf** clauses are both optional. You can have as many **ElseIf** statements as you want in a block **If**, but none can appear after the **Else** clause. Block **If** statements can be nested; that is, contained within one another.

What follows the **Then** keyword is examined to determine whether or not a statement is a block **If**. If anything other than a comment appears after **Then** on the same line, the statement is treated as a single-line **If** statement.

A block **If** statement must be the first statement on a line. The block **If** must end with an **End If** statement.

Requirements

Version 1

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

On Error Statement

Enables or disables error-handling.

```
On Error Resume Next
On Error GoTo 0
```

Remarks

If you don't use an **On Error Resume Next** statement anywhere in your code, any run-time error that occurs can cause an error message to be displayed and code execution stopped. However, the host running the code determines the exact behavior. The host can sometimes opt to handle such errors differently. In some cases, the script debugger may be invoked at the point of the error. In still other cases, there may be no apparent indication that any error occurred because the host does not to notify the user. Again, this is purely a function of how the host handles any errors that occur.

Within any particular procedure, an error is not necessarily fatal as long as error-handling is enabled somewhere along the call stack. If local error-handling is not enabled in a procedure and an error occurs, control is passed back through the call stack until a procedure with error-handling enabled is found and the error is handled at that point. If no procedure in the call stack is found to have error-handling enabled, an error message is displayed at that point and execution stops or the host handles the error as appropriate.

On Error Resume Next causes execution to continue with the statement immediately following the statement that caused the run-time error, or with the statement immediately following the most recent call out of the procedure containing the **On Error Resume Next** statement. This allows execution to continue despite a run-time error. You can then build the error-handling routine inline within the procedure.

An **On Error Resume Next** statement becomes inactive when another procedure is called, so you should execute an **On Error Resume Next** statement in each called routine if you want inline error handling within that routine. When a procedure is exited, the error-handling capability reverts to whatever error-handling was in place before entering the exited procedure.

Use **On Error GoTo 0** to disable error handling if you have previously enabled it using **On Error Resume Next**.

The following example illustrates use of the **On Error Resume Next** statement.

```
On Error Resume Next
Err.Raise 6   ' Raise an overflow error.
MsgBox "Error # " & CStr(Err.Number) & " " & Err.Description
Err.Clear   ' Clear the error.
```

Requirements

[Version 1]

See Also

Err Object | Exit Statement

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Option Explicit Statement

Forces explicit declaration of all variables in a script.

```
Option Explicit
```

Remarks

If used, the **Option Explicit** statement must appear in a script before any other statements.

When you use the **Option Explicit** statement, you must explicitly declare all variables using the **Dim**, **Private**, **Public**, or **ReDim** statements. If you attempt to use an undeclared variable name, an error occurs.

> **Tip** Use **Option Explicit** to avoid incorrectly typing the name of an existing variable or to avoid confusion in code where the scope of the variable is not clear.

The following example illustrates use of the **Option Explicit** statement.

```
Option Explicit   ' Force explicit variable declaration.
Dim MyVar   ' Declare variable.
MyInt = 10   ' Undeclared variable generates error.
MyVar = 10   ' Declared variable does not generate error.
```

Requirements

Version 1

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Private Statement

Declares private variables and allocates storage space. Declares, in a **Class** block, a private variable.

```
Private varname[([subscripts])][, varname[([subscripts])]] . . .
```

Arguments

varname
Name of the variable; follows standard variable naming conventions.

subscripts
Dimensions of an array variable; up to 60 multiple dimensions may be declared. The *subscripts* argument uses the following syntax:

upper [**,** upper] **. . .**

The lower bound of an array is always zero.

Remarks

Private statement variables are available only to the script in which they are declared.

A variable that refers to an object must be assigned an existing object using the **Set** statement before it can be used. Until it is assigned an object, the declared object variable is initialized as **Empty**.

You can also use the **Private** statement with empty parentheses to declare a dynamic array. After declaring a dynamic array, use the **ReDim** statement within a procedure to define the number of dimensions and elements in the array. If you try to redeclare a dimension for an array variable whose size was explicitly specified in a **Private**, Public, or **Dim** statement, an error occurs.

> **Note** When you use the **Private** statement in a procedure, you generally put the **Private** statement at the beginning of the procedure.

The following example illustrates use of the **Private** statement.

```
Private MyNumber   ' Private Variant variable.
Private MyArray(9)   ' Private array variable.
   ' Multiple Private declarations of Variant variables.
Private MyNumber, MyVar, YourNumber
```

Requirements

[Version 1]

See Also

[Dim Statement] | [Public Statement] | [ReDim Statement] | [Set Statement]

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Property Get Statement

Declares, in a **Class** block, the name, arguments, and code that form the body of a **Property** procedure that gets (returns) the value of a property.

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

Arguments

Public
: Indicates that the **Property Get** procedure is accessible to all other procedures in all scripts.

Default
: Used only with the **Public** keyword to indicate that the property defined in the **Property Get** procedure is the default property for the class.

Private
: Indicates that the **Property Get** procedure is accessible only to other procedures in the **Class** block where it's declared.

name
: Name of the **Property Get** procedure; follows standard variable naming conventions, except that the name can be the same as a **Property Let** or **Property Set** procedure in the same **Class** block.

arglist
: List of variables representing arguments that are passed to the **Property Get** procedure when it is called. Commas separate multiple arguments. The name of each argument in a **Property Get** procedure must be the same as the corresponding argument in a **Property Let** procedure (if one exists).

statements
: Any group of statements to be executed within the body of the **Property Get** procedure.

Set
: Keyword used when assigning an object as the return value of a **Property Get** procedure.

expression
: Return value of the **Property Get** procedure.

Remarks

If not explicitly specified using either **Public** or **Private**, **Property Get** procedures are public by default, that is, they are visible to all other procedures in your script. The value of local variables in a **Property Get** procedure is not preserved between calls to the procedure.

You can't define a **Property Get** procedure inside any other procedure (e.g. **Function** or **Property Let**).

The **Exit Property** statement causes an immediate exit from a **Property Get** procedure. Program execution continues with the statement that follows the statement that called the **Property Get** procedure. Any number of **Exit Property** statements can appear anywhere in a **Property Get** procedure.

Like a **Sub** and **Property Let** procedure, a **Property Get** procedure is a separate procedure that can take arguments, perform a series of statements, and change the value of its arguments. However, unlike a **Sub** and **Property Let**, you can use a **Property Get** procedure on the right side of an expression in the same way you use a **Function** or property name when you want to return the value of a property.

Requirements

Version 5

See Also

Class Statement | Dim Statement | Exit Statement | Function Statement | Private Statement | Property Let Statement | Property Set Statement | Public Statement | Set Statement | Sub Statement

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Property Let Statement

Declares, in a **Class** block, the name, arguments, and code that form the body of a **Property** procedure that assigns (sets) the value of a property.

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

```
   [statements]
End Property
```

Arguments

Public
: Indicates that the **Property Let** procedure is accessible to all other procedures in all scripts.

Private
: Indicates that the **Property Let** procedure is accessible only to other procedures in the **Class** block where it's declared.

name
: Name of the **Property Let** procedure; follows standard variable naming conventions, except that the name can be the same as a **Property Get** or **Property Set** procedure in the same **Class** block.

arglist
: List of variables representing arguments that are passed to the **Property Let** procedure when it is called. Commas separate multiple arguments. The name of each argument in a **Property Let** procedure must be the same as the corresponding argument in a **Property Get** procedure. In addition, the **Property Let** procedure will always have one more argument than its corresponding **Property Get** procedure. That argument is the value being assigned to the property.

value
: Variable to contain the value to be assigned to the property. When the procedure is called, this argument appears on the right side of the calling expression.

statements
: Any group of statements to be executed within the body of the **Property Let** procedure.

Remarks

If not explicitly specified using either **Public** or **Private**, **Property Let** procedures are public by default, that is, they are visible to all other procedures in your script. The value of local variables in a **Property Let** procedure is not preserved between calls to the procedure.

You can't define a **Property Let** procedure inside any other procedure (e.g. **Function** or **Property Get**).

The **Exit Property** statement causes an immediate exit from a **Property Let** procedure. Program execution continues with the statement that follows the statement that called the **Property Let** procedure. Any number of **Exit Property** statements can appear anywhere in a **Property Let** procedure.

> **Note** Every **Property Let** statement must define at least one argument for the procedure it defines. That argument (or the last argument if there is more than one) contains the actual value to be assigned to the property when the procedure defined by the **Property Let** statement is invoked. That argument is referred to as *value* in the preceding syntax.

Like a **Function** and **Property Get** procedure, a **Property Let** procedure is a separate procedure that can take arguments, perform a series of statements, and change the value of its arguments. However, unlike a **Function** and **Property Get** procedure, both of which return a value, you can only use a **Property Let** procedure on the left side of a property assignment expression.

Requirements

Version 5

See Also

Class Statement | Dim Statement | Exit Statement | Function Statement | Private Statement | Property Get Statement | Property Set Statement | Public Statement | Set Statement | Sub Statement

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Property Set Statement

Declares, in a **Class** block, the name, arguments, and code that form the body of a **Property** procedure that sets a reference to an object.

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

Arguments

Public
: Indicates that the **Property Set** procedure is accessible to all other procedures in all scripts.

Private
Indicates that the **Property Set** procedure is accessible only to other procedures in the **Class** block where it's declared.

name
Name of the **Property Set** procedure; follows standard variable naming conventions, except that the name can be the same as a **Property Get** or **Property Let** procedure in the same **Class** block.

arglist
List of variables representing arguments that are passed to the **Property Set** procedure when it is called. Commas separate multiple arguments. In addition, the **Property Set** procedure will always have one more argument than its corresponding **Property Get** procedure. That argument is the object being assigned to the property.

reference
Variable containing the object reference used on the right side of the object reference assignment.

statements
Any group of statements to be executed within the body of the **Property Set** procedure.

Remarks

If not explicitly specified using either **Public** or **Private**, **Property Set** procedures are public by default, that is, they are visible to all other procedures in your script. The value of local variables in a **Property Set** procedure is not preserved between calls to the procedure.

You can't define a **Property Set** procedure inside any other procedure (e.g. **Function** or **Property Let**).

The **Exit Property** statement causes an immediate exit from a **Property Set** procedure. Program execution continues with the statement that follows the statement that called the **Property Set** procedure. Any number of **Exit Property** statements can appear anywhere in a **Property Set** procedure.

> **Note** Every **Property Set** statement must define at least one argument for the procedure it defines. That argument (or the last argument if there is more than one) contains the actual object reference for the property when the procedure defined by the **Property Set** statement is invoked. That argument is referred to as *reference* in the preceding syntax.

Like a **Function** and **Property Get** procedure, a **Property Set** procedure is a separate procedure that can take arguments, perform a series of statements, and change the value of its arguments. However, unlike a **Function** and **Property Get** procedure, both of which return a value, you can only use a **Property Set** procedure on the left side of an object reference assignment (**Set** statement).

Requirements

Version 5

See Also

Class Statement | Dim Statement | Exit Statement | Function Statement | Private Statement | Property Get Statement | Property Let Statement | Public Statement | Set Statement | Sub Statement

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Public Statement

Declares public variables and allocates storage space. Declares, in a **Class** block, a private variable.

```
Public varname[([subscripts])][, varname[([subscripts])]] . . .
```

Arguments

varname
: Name of the variable; follows standard variable naming conventions.

subscripts
: Dimensions of an array variable; up to 60 multiple dimensions may be declared. The *subscripts* argument uses the following syntax:

  upper [, upper] . . .

  The lower bound of an array is always zero.

Remarks

Public statement variables are available to all procedures in all scripts.

A variable that refers to an object must be assigned an existing object using the **Set** statement before it can be used. Until it is assigned an object, the declared object variable is initialized as Empty.

You can also use the **Public** statement with empty parentheses to declare a dynamic array. After declaring a dynamic array, use the **ReDim** statement within a procedure to define the number of dimensions and elements in the array. If you try to redeclare a dimension for an array variable whose size was explicitly specified in a **Private**, **Public**, or **Dim** statement, an error occurs.

The following example illustrates the use of the **Public** statement:

```
Public MyNumber   ' Public Variant variable.
Public MyArray(9)   ' Public array variable.
   ' Multiple Public declarations of Variant variables.
Public MyNumber, MyVar, YourNumber
```

Requirements

Version 1

See Also

Dim Statement | Private Statement | ReDim Statement | Set Statement

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Randomize Statement

Initializes the random-number generator.

Randomize [*number*]

The *number* argument can be any valid numeric expression.

Remarks

Randomize uses *number* to initialize the **Rnd** function's random-number generator, giving it a new seed value. If you omit *number,* the value returned by the system timer is used as the new seed value.

If **Randomize** is not used, the **Rnd** function (with no arguments) uses the same number as a seed the first time it is called, and thereafter uses the last generated number as a seed value.

> **Note** To repeat sequences of random numbers, call **Rnd** with a negative argument immediately before using **Randomize** with a numeric argument. Using **Randomize** with the same value for *number* does not repeat the previous sequence.

The following example illustrates use of the **Randomize** statement.

```
Dim MyValue, Response
Randomize   ' Initialize random-number generator.
Do Until Response = vbNo
   MyValue = Int((6 * Rnd) + 1)   ' Generate random value between 1 and 6.
   MsgBox MyValue
   Response = MsgBox ("Roll again? ", vbYesNo)
Loop
```

Requirements

Version 1

See Also

Rnd Function | Timer Function

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

ReDim Statement

Declares dynamic-array variables, and allocates or reallocates storage space at procedure level.

```
ReDim [Preserve] varname(subscripts) [, varname(subscripts)] . . .
```

Arguments

Preserve
Preserves the data in an existing array when you change the size of the last dimension.

varname
Name of the variable; follows standard variable naming conventions.

subscripts
Dimensions of an array variable; up to 60 multiple dimensions may be declared. The *subscripts* argument uses the following syntax:

upper [**,**upper] **. . .**

The lower bound of an array is always zero.

Remarks

The **ReDim** statement is used to size or resize a dynamic array that has already been formally declared using a **Private**, **Public**, or **Dim** statement with empty parentheses (without dimension subscripts). You can use the **ReDim** statement repeatedly to change the number of elements and dimensions in an array.

If you use the **Preserve** keyword, you can resize only the last array dimension, and you can't change the number of dimensions at all. For example, if your array has only one dimension, you can resize that dimension because it is the last and only dimension. However, if your array has two or more dimensions, you can change the size of only the last dimension and still preserve the contents of the array.

The following example shows how you can increase the size of the last dimension of a dynamic array without erasing any existing data contained in the array.

```
ReDim X(10, 10, 10)
. . .
```

```
ReDim Preserve X(10, 10, 15)
```

Caution If you make an array smaller than it was originally, data in the eliminated elements is lost.

When variables are initialized, a numeric variable is initialized to 0 and a string variable is initialized to a zero-length string (""). A variable that refers to an object must be assigned an existing object using the **Set** statement before it can be used. Until it is assigned an object, the declared object variable has the special value **Nothing**.

Requirements

Version 1

See Also

Dim Statement | Set Statement

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Rem Statement

Includes explanatory remarks in a program.

```
Rem comment
```

-or-

```
' comment
```

The *comment* argument is the text of any comment you want to include. After the **Rem** keyword, a space is required before *comment*.

Remarks

As shown in the syntax section, you can use an apostrophe (') instead of the **Rem** keyword. If the **Rem** keyword follows other statements on a line, it must be separated from the statements by a colon. However, when you use an apostrophe, the colon is not required after other statements.

The following example illustrates the use of the **Rem** statement.

```
Dim MyStr1, MyStr2
MyStr1 = "Hello" : Rem Comment after a statement separated by a colon.
MyStr2 = "Goodbye" ' This is also a comment; no colon is needed.
Rem Comment on a line with no code; no colon is needed.
```

Requirements

Version 1

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Select Case Statement

Executes one of several groups of statements, depending on the value of an expression.

```
Select Case testexpression
   [Case expressionlist-n
      [statements-n]] . . .
   [Case Else expressionlist-n
      [elsestatements-n]]
End Select
```

Arguments

testexpression
Any numeric or string expression.
expressionlist-n
Required if **Case** appears. Delimited list of one or more expressions.
statements-n
One or more statements executed if *testexpression* matches any part of *expressionlist-n*.
elsestatements-n
One or more statements executed if *testexpression* doesn't match any of the **Case** clauses.

Remarks

If *testexpression* matches any **Case** *expressionlist* expression, the statements following that **Case** clause are executed up to the next **Case** clause, or for the last clause, up to **End Select**. Control then passes to the statement following **End Select**. If *testexpression* matches an *expressionlist* expression in more than one **Case** clause, only the statements following the first match are executed.

The **Case Else** clause is used to indicate the *elsestatements* to be executed if no match is found between the *testexpression* and an *expressionlist* in any of the other **Case** selections. Although not required, it is a good idea to have a **Case Else** statement in your **Select Case** block to handle unforeseen *testexpression* values. If no **Case** *expressionlist* matches *testexpression* and there is no **Case Else** statement, execution continues at the statement following **End Select**.

Select Case statements can be nested. Each nested **Select Case** statement must have a matching **End Select** statement.

The following example illustrates the use of the **Select Case** statement.

```
Dim Color, MyVar
Sub ChangeBackground (Color)
   MyVar = lcase (Color)
   Select Case MyVar
      Case "red"     document.bgColor = "red"
      Case "green"   document.bgColor = "green"
      Case "blue"    document.bgColor = "blue"
      Case Else      MsgBox "pick another color"
   End Select
End Sub
```

Requirements

Version 1

See Also

If...Then...Else Statement

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Set Statement

Assigns an object reference to a variable or property, or associates a procedure reference with an event.

Set *objectvar* = {*objectexpression* | **New** *classname* | **Nothing**}

-or-

Set *object.eventname* = **GetRef**(*procname*)

Parameters

objectvar
: Required. Name of the variable or property; follows standard variable naming conventions.

objectexpression
: Optional. Expression consisting of the name of an object, another declared variable of the same object type, or a function or method that returns an object of the same object type.

New
: Keyword used to create a new instance of a class. If *objectvar* contained a reference to an object, that reference is released when the new one is assigned. The **New** keyword can only be used to create an instance of a class.

classname

Optional. Name of the class being created. A class and its members are defined using the **Class** statement.

Nothing

Optional. Discontinues association of *objectvar* with any specific object or class. Assigning *objectvar* to **Nothing** releases all the system and memory resources associated with the previously referenced object when no other variable refers to it.

object

Required. Name of the object with which *event* is associated.

event

Required. Name of the event to which the function is to be bound.

procname

Required. String containing the name of the **Sub** or **Function** being associated with the *event*.

Remarks

To be valid, *objectvar* must be an object type consistent with the object being assigned to it.

The **Dim**, **Private**, **Public**, or **ReDim** statements only declare a variable that refers to an object. No actual object is referred to until you use the **Set** statement to assign a specific object.

Generally, when you use **Set** to assign an object reference to a variable, no copy of the object is created for that variable. Instead, a reference to the object is created. More than one object variable can refer to the same object. Because these variables are references to (rather than copies of) the object, any change in the object is reflected in all variables that refer to it.

```
Function ShowFreeSpace(drvPath)
   Dim fso, d, s
   Set fso = CreateObject("Scripting.FileSystemObject")
   Set d = fso.GetDrive(fso.GetDriveName(drvPath))
   s = "Drive " & UCase(drvPath) & " - "
   s = s & d.VolumeName  & "<BR>"
   s = s & "Free Space: " & FormatNumber(d.FreeSpace/1024, 0)
   s = s & " Kbytes"
   ShowFreeSpace = s
End Function
```

Using the **New** keyword allows you to concurrently create an instance of a class and assign it to an object reference variable. The variable to which the instance of the class is being assigned must already have been declared with the **Dim** (or equivalent) statement.

Refer to the documentation for the **GetRef** function for information on using **Set** to associate a procedure with an event.

Requirements

Version 1

See Also

= Operator | Dim Statement | GetRef Function | ReDim Statement

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

Sub Statement

Declares the name, arguments, and code that form the body of a **Sub** procedure.

```
[Public [Default] | Private] Sub name [(arglist)]
   [statements]
   [Exit Sub]
   [statements]
End Sub
```

Arguments

Public
: Indicates that the **Sub** procedure is accessible to all other procedures in all scripts.

Default
: Used only with the **Public** keyword in a **Class** block to indicate that the **Sub** procedure is the default method for the class. An error occurs if more than one **Default** procedure is specified in a class.

Private
: Indicates that the **Sub** procedure is accessible only to other procedures in the script where it is declared.

name
Name of the **Sub**; follows standard variable naming conventions.
arglist
List of variables representing arguments that are passed to the **Sub** procedure when it is called. Commas separate multiple variables.
statements
Any group of statements to be executed within the body of the **Sub** procedure.

The *arglist* argument has the following syntax and parts:

```
[ByVal | ByRef] varname[( )]
```

Arguments

ByVal
Indicates that the argument is passed by value.
ByRef
Indicates that the argument is passed by reference.
varname
Name of the variable representing the argument; follows standard variable naming conventions.

Remarks

If not explicitly specified using either **Public** or **Private**, **Sub** procedures are public by default, that is, they are visible to all other procedures in your script. The value of local variables in a **Sub** procedure is not preserved between calls to the procedure.

You can't define a **Sub** procedure inside any other procedure (e.g. **Function** or **Property Get**).

The **Exit Sub** statement causes an immediate exit from a **Sub** procedure. Program execution continues with the statement that follows the statement that called the **Sub** procedure. Any number of **Exit Sub** statements can appear anywhere in a **Sub** procedure.

Like a **Function** procedure, a **Sub** procedure is a separate procedure that can take arguments, perform a series of statements, and change the value of its arguments. However, unlike a **Function** procedure, which returns a value, a **Sub** procedure can't be used in an expression.

You call a **Sub** procedure using the procedure name followed by the argument list. See the **Call** statement for specific information on how to call **Sub** procedures.

Caution **Sub** procedures can be recursive, that is, they can call themselves to perform a given task. However, recursion can lead to stack overflow.

Variables used in **Sub** procedures fall into two categories: those that are explicitly declared within the procedure and those that are not. Variables that are explicitly declared in a procedure (using **Dim** or the equivalent) are always local to the procedure. Variables that are used but not explicitly declared in a procedure are also local, unless they are explicitly declared at some higher level outside the procedure.

> **Caution** A procedure can use a variable that is not explicitly declared in the procedure, but a naming conflict can occur if anything you have defined at the script level has the same name. If your procedure refers to an undeclared variable that has the same name as another procedure, constant or variable, it is assumed that your procedure is referring to that script-level name. To avoid this kind of conflict, use an **Option Explicit** statement to force explicit declaration of variables.

Requirements

Version 1

See Also

Call Statement | Dim Statement | Exit Statement | Function Statement

---

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

While...Wend Statement

Executes a series of statements as long as a given condition is **True**.

```
While condition
   Version [statements]
Wend
```

Arguments

condition
: Numeric or string expression that evaluates to **True** or **False**. If *condition* is Null, *condition* is treated as **False**.

statements
: One or more statements executed while condition is **True**.

Remarks

If *condition* is **True**, all statements in *statements* are executed until the **Wend** statement is encountered. Control then returns to the **While** statement and *condition* is again checked. If *condition* is still **True**, the process is repeated. If it is not **True**, execution resumes with the statement following the **Wend** statement.

While...Wend loops can be nested to any level. Each **Wend** matches the most recent **While**.

> **Note** The **Do...Loop** statement provides a more structured and flexible way to perform looping.

The following example illustrates use of the **While...Wend** statement:

```
Dim Counter
Counter = 0   ' Initialize variable.
While Counter < 20   ' Test value of Counter.
   Counter = Counter + 1   ' Increment Counter.
   Alert Counter
Wend   ' End While loop when Counter > 19.
```

Requirements

Version 1

See Also

Do...Loop Statement

Build: Topic Version 5.6.9309.1546

Visual Basic Scripting Edition

With Statement

Executes a series of statements on a single object.

```
With object
   statements
End With
```

Arguments

object
Required. Name of an object or a function that returns an object.
statements
Required. One or more statements to be executed on *object*.

Remarks

The **With** statement allows you to perform a series of statements on a specified object without requalifying the name of the object. For example, to change a number of different properties on a single object, place the property assignment statements within the **With** control structure, referring to the object once instead of referring to it with each property assignment. The following example illustrates use of the **With** statement to assign values to several properties of the same object.

```
With MyLabel
   .Height = 2000
   .Width = 2000
   .Caption = "This is MyLabel"
End With
```

While property manipulation is an important aspect of **With** functionality, it is not the only use. Any legal code can be used within a **With** block.

> **Note** Once a **With** block is entered, *object* can't be changed. As a result, you can't use a single **With** statement to affect a number of different objects.

You can nest **With** statements by placing one **With** block within another. However, because members of outer **With** blocks are masked within the inner **With** blocks, you must provide a fully qualified object reference in an inner **With** block to any member of an object in an outer **With** block.

Important Do not jump into or out of **With** blocks. If statements in a **With** block are executed, but either the **With** or **End With** statement is not executed, you may get errors or unpredictable behavior.

Requirements

Version 1

See Also

Set Statement

Build: Topic Version 5.6.9309.1546