Using VBScript

with

InduSoft Web Studio

Table of Contents

About VBScript

Visual Basic Script Language (VBScript) is one of Microsoft's scripting languages that is commonly associated with Server-side and Client-side web applications. However, Microsoft has opened up VBScript to developers and now VBScript can be found in a variety of applications. InduSoft has standardized on VBScript since it provides a significant subset of Microsoft Visual Basic's functionality, and VBScript supports all of Microsoft's operating system platforms including Windows CE, unlike VBA (Visual Basic for Applications) which cannot support the Windows CE runtime environment.

VBScript is a programming language that is often viewed as a dialect of VBA (Visual Basic for Applications), although it is really its own language. The VBScript language attempts to balance flexibility, capability and ease of use. VBA is a subset of Visual Basic that was developed to automate Microsoft Office applications, whereas VBScript was originally developed to support Server-side and Client-side web applications. Although VBScript and VBA provide many of the same features, there are some differences between them, primarily due to the applications they were each developed to support.

So before we get into details of the VBScripting language, perhaps it is worthwhile to review how VBScript is used with InduSoft Web Studio (IWS) or alternatively, why VBScript is included with IWS. IWS provides an easy-to-use development environment that configures predefined objects to support an HMI/SCADA application. Applications can be built quickly and are relatively easy to support, even by someone other than the original developer. By comparison, programming languages such as Visual Basic can be used to develop an HMI/SCADA application, but the lower per-copy licensing cost savings quickly gets offset by much higher development costs and support costs. A programming development environment is clearly more flexible than a configuration development environment, but there is a significant cost associated with programming that makes it an unattractive alternative for HMI/SCADA applications. By adding VBScript support to IWS, InduSoft lets you chose between configuration and programming to meet your application needs and develop applications efficiently.

Using VBScript With InduSoft HMI/SCADA Applications

InduSoft Web Studio (IWS) supports both a simple, proprietary scripting language (worksheet style) using one or more Math worksheets, as well as VBScript (new with IWS Version 6.1). Developers can use either scripting language or a combination of both. VBScript code is placed in one of several modules, based on the functionality to be performed and the scope of the code and its variables. This subject is covered more completely in the *VBScript Configuration and Operation in IWS* section.

Examples of how VBScript can be used:

- To execute a logic sequence or a routine when opening or closing a screen, or while the screen is open
- To execute a logic sequence in the background
- Run a simple VBScipt code segment based on an IWS object's command dynamic
- Interaction with IWS Tags and control of IWS built-in functions
- Manipulation of ActiveX Controls and ActiveX Control event handler
- Simple file I/O (e.g. text files)
- Database interfaces (e.g. via ADO.NET), especially where use of SQL is required
- Interface to Windows Management Instrumentation (WMI) and Web Services (via WSDL)
- Interface to Microsoft Office applications (e.g. Excel, Access, Word) and Microsoft Office components via OLE Automation
- Run on a Web Thin Client

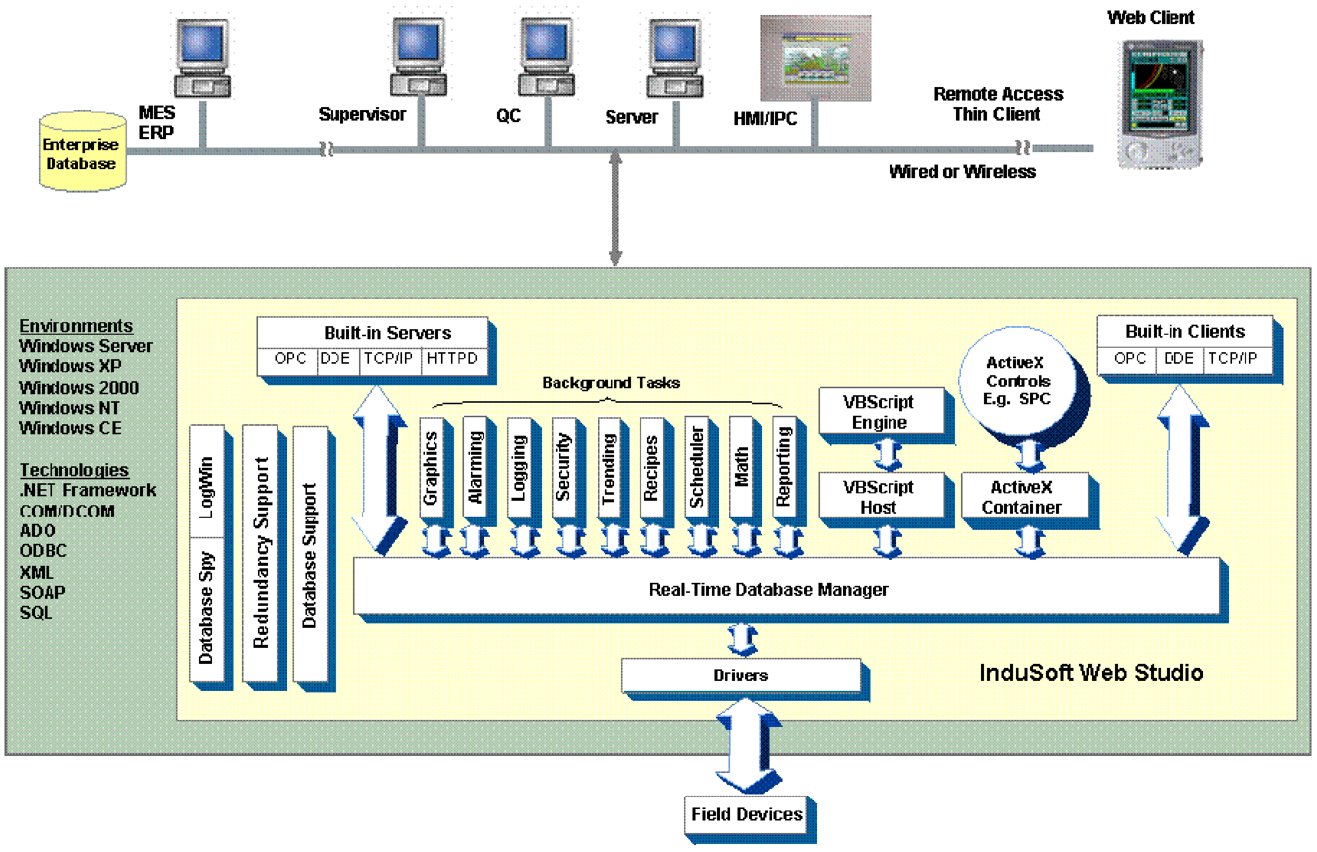
Where you should use IWS instead of VBScript

- User Interface. IWS does not support Windows Scripting, which typically provides the User Interface for VBScript via Forms.
- Device I/O (e.g. PLC communications). VBScript does not directly support serial or network communications.

IWS implements Visual Basic Script Edition 5.5 or higher, and functions as the “host” for VBScript. IWS provides an integrated development environment where the HMI/SCADA application developer can take advantage of the functionality and ease of use of VBScript, yet have access to all IWS tags and all built-in functions directly from VBScript. The diagram below illustrates the IWS architecture. Since VBScript is an interpreted language, the VBScript Engine parses the language at runtime and executes commands subject to limitations placed by the VBScript Host. InduSoft allows VBScript code to be located several areas in an IWS application:

- **Global Procedures**. This is an area for subroutines and functions that can be called by any other VBScript routine, or by a built-in IWS function (requires IWS Version 6.1 Service Pack 1 or later).
- **Graphic Script**. Code in this area gets executed whenever any graphics (screens) are active.
- **Screen Script**. This is where code is executed when an individual screen is active.
- **Command Dynamic**. When an object has a Command Dynamic, one option is to run VBScript code.
- **ActiveX Events**. A VBScript code segment can be run based on an ActiveX event
- **Background Task**. VBScript code can be running as a background task. One or more VBScript groups are supported, allowing conditional processing of the various VBScript background tasks.

This subject is covered more completely later in the *VBScript Configuration and Operation in IWS* section.



InduSoft Web Studio Version 6.1 Internal Architecture

In a Web Thin Client configuration, VBScripts associated with a screen can run either on the workstation runtime display or on a Web Thin Client station running Microsoft Internet Explorer. The VBScript routines that can execute on a Web Thin Client include those located in a Screen Script, a Command Dynamic, and an ActiveX Event. Since VBScript runs on all Microsoft operating system platforms, there are no limitations to VBScript running on any Microsoft compatible platform.

VBScript Limitations in IWS

Microsoft initially developed VBScript to work with websites (web pages). In the web server environment, VBScript was designed to work with the Windows Scripting host and ASP, which provide file access and form generation. On the web client side, VBScript was designed to work with Microsoft Internet Explorer using HTML and DHTML, which provide display generation. So as a result of the initial design goals, VBScript does not have much in the way of built-in language support for Forms, File I/O, Communications or direct Printing control. Additionally, IWS has its own built-in web server and does not use ASP.

By using IWS built-in functions, ActiveX controls and Microsoft Office Applications (or components), there are several methods for workarounds to these limitations as well as to extend VBScript's capability.

The following are some of VBScript's limitations and workarounds.

Item	VBScript	Workarounds
Forms	Does not support	Use IWS objects for user interface, pass parameters to IWS. Can also use ActiveX Controls.
File I/O	Limited support directly	Use Scripting Objects and/or IWS built-in functions. Can also use ActiveX Controls.
Communications	Does not directly support	Use IWS built-in functions or 3rd party ActiveX controls
Printing	Does not directly support	Use Microsoft Office Applications or IWS built-in functions
Charting/Graphing	Does not directly support	Use IWS trending, Microsoft Office Applications, Microsoft Office Components, or 3rd party ActiveX controls
DDE	Does not support	Supported in IWS built-in commands (not under Windows CE).

The Microsoft Visual Basic Family

VBScript is part of a family of Microsoft programming languages that support object-oriented programming. This family of products is derived from the Basic programming language, first developed in 1964. Once study recently indicated that over 50% of all programmers are familiar with VB (Visual Basic) programming.

Basic, VB (Visual Basic), VB.NET, VBA and VBScript – The Evolution

Most everyone is familiar with Basic, the Beginner's All-purpose Symbolic Instruction Code that has been around since 1964. Originally designed to teach non-science students about computers, it was one of the first high-level programming languages ported to the PC in the 1980's. It has continued to evolve with programming and operating system technology. Here is a quick summary of the different versions today:

- Basic — A simple high-level programming language developed in 1964. Migrated to the PC platform in the 1980's, with many versions developed.
- VB — Visual Basic. An event-driven programming version of Basic, supporting graphical user interfaces (GUI), database access and ActiveX controls that was introduced in 1991. VB Version 6 was the last version released (1998).
- VB.NET — The successor to VB launched in 2002. Supports Microsoft .NET framework architecture and is a true object-oriented programming language.
- VBA — Visual Basic for Applications is a version of VB (most compatible with Version 6) that is built into Microsoft Office products (Word, Excel, Access, Outlook, PowerPoint) and into some other 3rd party products. Unlike VB or VB.NET, VBA does not run stand-alone and only runs from a host application, usually within a Microsoft Office application. VBA can control an second application while running in a host application. VBA works on Windows XP/2000/NT platforms only.
- VBScript — VBScript is considered a dialect of VBA and is the default language for website Active Server Pages (ASP). Like VBA, VBScript does not run stand-alone and only runs from a host application. It is run by the operating system's Windows Script Host and can be used for Server-side Windows scripting or Client-side Web Page scripting using Microsoft Internet Explorer. A key advantage of VBScript is that it is supported under Windows CE.

Differences between VBScript and VBA

Since other HMI/SCADA products support VBA, it might be worth highlighting some of the key differences between VBScript and VBA. For HMI/SCADA applications, these differences are relatively minor. However, VBScript support for the Windows CE operating system is a major differentiator between the two products. For additional details or a complete listing of the differences, please reference the MSDN website at http://msdn.microsoft.com.

Key differences between VBScript vs. VBA

Item	VBA	VBScript
Primary Purpose	Automation of MS Office Applications	Automation of Web Services
Support for Windows CE	No	Yes
Data Types	Stronger Type Declaration. Many data types supported. (e.g. String, Integer, Date, Boolean)	Typeless, uses Variant Type. The final data subtype will be determined at runtime based on use. Supports same data subtypes as VBA and VB (e.g. String, Integer, Date, Boolean, etc)
Dimension Statement	Dim Var as Type	Dim Var (Cannot specify Type, but it is determined at runtime based on use)
Class Block declaration	Must use separate Class Module	Class Block Declaration supported
Object	Clipboard Collection	Not supported
Object Manipulation	TypeOf	Not supported
Eval function	Not supported	Expression evaluation supported
Execute function	Not supported	Allows interpreted code to be executed on the fly.
RegExp	No	Allows creation of regular expressions
Error Handling	Several different types	Supported but more limited
Arrays	Lower bound can be <>0	Lower bound is 0
File I/O	Supported	Not directly supported but VBScript can use FileSystemObject and can access IWS built-in I/O functions
DDE	Supported	Not supported
Financial functions	Supported	Not supported
Strings	Fixed length strings	Variable length only
Debugging	Debug, Print, End, Stop	Use MsgBox or IWS built-in functions
Line labels	Supported	Not supported

VBScript Hosting Environments

While much of the material contained in the document covers the VBScript language and its use for IWS applications, it is important to understand conceptually how VBScript works in an IWS environment. If you browse the web for information on VBScript, you will likely find a plethora of information, but many of the examples are for running VBScript with ASP using the Windows Scripting Host.

VBScript was developed using a Microsoft technology called ActiveX scripting, which is a COM-based specification that allows the development of runtime engines for virtually any scripting language. Other scripting languages include JScript. VBScript can create an instance, or instantiate, a COM object, and thus through VBScript, many system features can be controlled such as ActiveX Controls, the FIleSystemObject (providing access to the Windows file system), Microsoft Office Automation (COM), and ActiveX Database Objects (ADO).

The VBScript Scripting Engine runs on a host, and there are several hosts that can run VBScript (or any ActiveX Scripting-compliant engines) including Windows Scripting Host and Microsoft Internet Explorer. VBScript can be used in conjunction with Windows Scripting Host (WSH) to automate system administration tasks. WSH is part of the Microsoft operating system and treats a VBScript application like a powerful batch file. VBScript applications can also be found with Web-based shell views. Most frequently, VBScript is used with Active Server Pages (ASP) for Server-side web applications and Microsoft Internet Explorer for Client-side web applications.

Stating with Version 6.1, IWS is now a host for the VBScript Scripting Engine. When used in conjunction with IWS, IWS becomes the only host for the VBScript Scripting Engine that is used. WSH is not used by IWS, even though WSH may be resident on the PC running the IWS application.

InduSoft has placed implemented VBScript host environment in a manner that is logically consistent with the current IWS application development environment and licensing method. What this means is that there are multiple locations in the development environment where VBScript code segments can be located (so the code is located close to its use), and restrictions placed on the scope of procedures and variables. In IWS, there is no such thing as a Global Variable that is accessible by any VBScript code segment. The IWS tags and built-in procedures can be accessed by any VBScript code segment. The restrictions and interaction with IWS tags and built-in functions implemented by the IWS VBScript Host are covered in more detail in the **VB Configuration and Operation in IWS** section.

The VBScript Scripting Engine performs a few key functions. It performs syntax checking in the development environment (e.g. right mouse click on a VBScript Interface, then select Check Script). It also interacts with IntelliSense, an auto-completion tool that provides reference to available functions (VBScript and IWS), IWS tags and ActiveX Controls (name, Properties and Methods). And most importantly, it executes the VBScript code at runtime, providing error messages if an error occurs. It should be noted that unlike most programming languages, VBScript is not compiled; it runs in an interpreted mode. The VBScript Scripting Engine (vbscript.dll) is responsible for interpreting (via the VBScript Parser, a part of the VBScript Scripting Engine) and executing the VBScript statements, and it does so quite efficiently. IWS uses Version 5.6 or later of the Microsoft VBScript Scripting Engine.

There are no limitations on the number of VBScript variables supported in IWS, however the amount of storage for VBScript variables is determined by the amount of memory available in your system.
VBScript variables do not count against IWS tag limits for licensing purposes.

VBScript Language Overview

This section contains a short summary of the VBScript Language. A more complete reference of the VBScript language can be found in the Appendix at the end of these materials.

VBScript Functionality

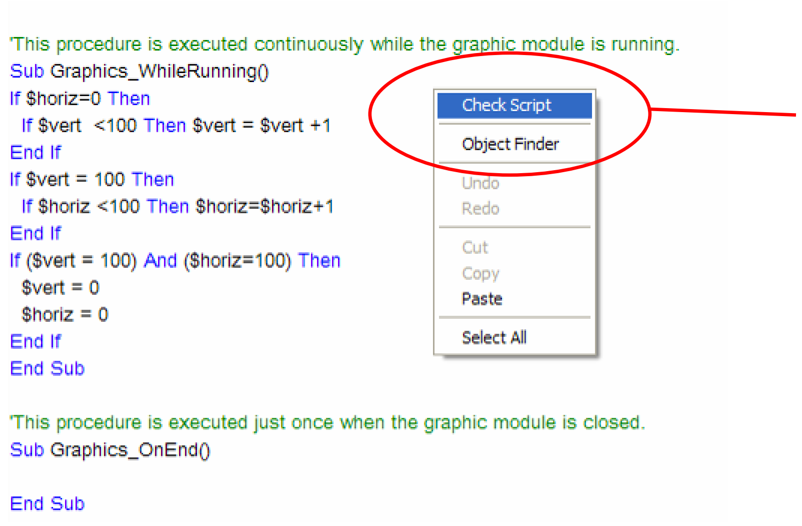
VBScript has inherited much of VB & VBA's functionality including support for math operations, string manipulation, arrays, flow control, data conversion, procedures, COM objects, and date/time functions. Since VBScript was initially designed for Web applications, direct support for file I/O and user interface functions was not included. However, VBScript can use the *FileSystemObject* COM object (scrrun.dll) to manipulate local files and folders.

VBScript does not support explicitly declared data types. This was eliminated to speed up the runtime performance of the VBScript Scripting Engine. All variables are type *Variant* and their subtype (e.g. Integer, Real, etc.) is determined at runtime.

VBScript Elements

There are several VBScript elements, but the most important ones are variables, constants and types. A variable is an item holding data that can change during the execution of the VBScript program. A constant is an item that holds data but cannot change during the execution of the VBScript program. The data that variables and constants hold can be classified into types.

Note that with IWS, you can check the VBScript syntax for errors by choosing the Check VBScript command (right mouse click when in a VBScript interface). VBScript is always checked when saving the Script interface.



```
'This procedure is executed continuously while the graphic module is running.
Sub Graphics_WhileRunning()
If $horiz=0 Then
  If $vert  <100 Then $vert = $vert +1
End If
If $vert = 100 Then
  If $horiz <100 Then $horiz=$horiz+1
End If
If ($vert = 100) And ($horiz=100) Then
  $vert = 0
  $horiz = 0
End If
End Sub

'This procedure is executed just once when the graphic module is closed.
Sub Graphics_OnEnd()

End Sub
```

The Check Script function can be invoked following a right mouse click when the cursor is on the VBScript Interface. Note that Comments are in Green, VBScript Functions and KeyWords are in Blue, Variables are in Black

The VBScript elements that are covered in this material (and the Appendix) include:

- Variables (Type, Declaration, Scope)
- Constants (Explicit, Implicit)
- Keywords
- Errors (Runtime, Syntax)
- Operators
- Functions and Procedures
- Statements
- Objects and Collections
- Example VBScript Applications

Variable Data Types and Subtypes

All variables in VBScript are a data type called **Variant**. This means that you do not (and cannot) explicitly declare the variable type. In fact, with VBScript you do not need the Dim statement to allocate storage for a variable. At runtime, the Parser in the VBScript Scripting Engine determines the **Variant** data subtype to be used. These correspond to the more traditional classifications of data types (see chart below).

Variant data subtypes

Subtype	Description
Boolean	Either *True* or *False*
Byte	Contains integer in the range 0 to 255
Currency	Floating-point number in the range -922,337,203,685,477.5808 to 922,337,203,685,477.5807
Date(Time)	Contains a number that represents a date between January 1, 100 to December 31, 9999
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
Empty	Uninitialized Variant
Error	Contains an error number used with runtime errors
Integer	Contains integer in the range -32,768 to 32,767
Long	Contains integer in the range -2,147,483,648 to 2,147,483,647
Null	A variant containing no valid data
Object	Contains an object reference
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
String	Contains a variable-length string that can be up to approximately 2 billion characters in length.

The Parsers choice of data subtype will depend on how the variable is used in a statement or function. Note that a variable's subtype can change within a code segment.

Data Subtype Identification

If it is important to determine the **Variant** data subtype used at runtime, you may use any of the three categories of functions to determine the data subtype:

- The **VarType**(*variable*) function which returns a code based on the **Variant** data subtype used
- Various **IsXxxx**(*variable*) functions which return boolean values indicating whether the variable is of a specific data subtype.
- A **TypeName**(*variable*) function which returns a string based indicating the data subtype

Example:

```
If varType(a) = vbInteger Then
     Msgbox "a is an Integer"
EndIf
```

Data Subtype Conversion

VBScript provides several functions that convert a variable from one data subtype to another. Since VBScript uses the **Variant** data type, these functions are not generally required. However, when passing data between IWS (or CEView) and VBScipt, or calling built-in IWS functions from VBScript where variables need to be put into the proper argument format, these VBScript data subtype conversion functions can be very useful.

Example:

```
a = 4.2
b = cInt (a)                    ' b is an Integer with a value of 4
```

Variable Naming Rules & Conventions

VBScript has four primary rules for naming. These are:

1. Variable names must begin with an alpha character (a..z, A...Z) or an underscore character
2. After the first character, the variable name can contain letters, digits and underscores
3. Variable names must be less than 255 characters in length
4. The variable name must be unique in the scope in which they are declared

VBScript variable names are not case sensitive. Microsoft recommends following their naming convention for variables, which puts attaches different prefixes to the variable name based on the data subtype.

Variable Scope

Variables have “scope” which defines a variable’s visibility or accessibility from one procedure (or VBScript Interface) to another, which is principally determined by where you declare the variable. Generally, when you declare a variable within a procedure, only code within that procedure can access or change the value of that variable. This is called local scope and is for 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. However, as previously noted, InduSoft enforces certain restrictions on the scope of Variables and Procedures.

VBScript Constants

VBScript supports both *explicit* and *implicit* constants. Constants should never be used as variable names.

Explicit constants are defined by the programmer. Explicit constants have a defined value which, unlike a variable, is not allowed to change during the life of the script.

Implicit constants are pre-defined by VBScript. VBScript implicit constants usually begin with a **vb** prefix. VBScript implicit constants are available to the VBScript programmer without having to define them. Other objects, such as those used by ADO.NET, also have implicit constants predefined, usually with different prefixes. However, the implicit constants for these objects may not be know to VBScript and if not, will have to be defined as an explicit constant.

VBScript defines the following categories of implicit Constants:

Intrinsic Constant Category	Intrinsic Constant Category
Color Constants	File Attribute Constants
Comparison Constants	File Input/Output Constants
Date and Time Constants	MsgBox Constants
Date Format Constants	MsgBox Function Constants
Days of Week Constants	SpecialFolder Constants
New Years Week Constants	String Constants
Error Constants	Tristate Constants
VBScript Runtime Errors	VarType Constants
VBScript Syntax Errors	Locale ID (LCID)

Declaring VBScript Variables and Constants

VBScript does not require the explicit declaration of scalar variables, i.e. those variables with only one value assigned at any given time. Arrays, Objects (except **Err**) and Constants must be declared. While it may initially be convenient not to declare variables, any typing (spelling) errors of the variable or constant names may produce unexpected results at runtime.

VBScript Keywords

VBScript has many keywords. Keywords are merely the names or symbols used with built-in VBScript functions. Keywords are reserved, i.e. they may not be used by the programmer as names of variables or constants. VBScript keywords can be grouped into categories which include:

- Constants & Literals
- Operators
- Functions
- Statements
- Objects

Operators

VBScript defines various operators that perform operations based on the **Variant** subdata type(s). Arithmetic operators are used to perform operations on two or more numbers.

Arithmetic

Symbol	Definition
+	Add
-	Subtract
*****	Multiply
/	Divide
\	Integer Divide
^	Exponentiation
MOD	Modulus Division

Comparison

Symbol	Definition
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
=	Equal or assignment
<>	Not equal

Logical

Symbol	Definition
AND	And
OR, |	Or
XOR	Exclusive OR
Eqv	Equivalence
Imp	Implication
Not	NOT

String

Symbol	Definition
&, +	Concatenation

Object

Symbol	Definition
Is	Is (compare)

IWS

Symbol	Definition
$	Access to IWS Tags and Built-in functions

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:

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 its precedence does fall in 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 only checks to determine if two object references refer to the same object.

Operator 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	&

Functions

VBScript contains a number of built-in functions (not to be confused with the Function Procedure). These functions may or may not have arguments. These functions are called in a statement and may return a result that can be assigned to a variable. VBScript's functions are grouped as follows:

Array Functions

Array Functions	Description
Array	Returns a variant containing an array
Filter	Returns a zero-based array that contains a subset of a string array based on a filter criteria
IsArray	Returns a Boolean value that indicates whether a specified variable is an array
Join	Returns a string that consists of a number of substrings in an array
LBound	Returns the smallest subscript for the indicated dimension of an array
Split	Returns a zero-based, one-dimensional array that contains a specified number of substrings
UBound	Returns the largest subscript for the indicated dimension of an array

Data Conversion Functions

Function	Description
Abs	Returns the absolute value of a specified number
Asc	Converts the first letter in a string to its ASCII decimal representation
CBool	Converts an expression to a variant of subtype Boolean
CByte	Converts an expression to a variant of subtype Byte
CCur	Converts an expression to a variant of subtype Currency
CDate	Converts a valid date and time expression to the variant of subtype Date
CDbl	Converts an expression to a variant of subtype Double
Chr	Converts the specified ANSI code to a character
CInt	Converts an expression to a variant of subtype Integer
CLng	Converts an expression to a variant of subtype Long
CSng	Converts an expression to a variant of subtype Single
CStr	Converts an expression to a variant of subtype String
Fix	Returns the integer part of a specified number
Hex	Returns the hexadecimal value of a specified number
Int	Returns the integer part of a specified number
Oct	Returns the octal value of a specified number
Round	Returns a rounded number
Sgn	Returns the integer portion of a number

Date and Time Functions

Function	Description
CDate	Converts a valid date and time expression to the variant of subtype Date
Date	Returns the current system date
DateAdd	Returns a date to which a specified time interval has been added
DateDiff	Returns the number of intervals between two dates
DatePart	Returns the specified part of a given date
DateSerial	Returns the date for a specified year, month, and day
DateValue	Returns a date
Day	Returns a number that represents the day of the month (between 1 and 31, inclusive)
FormatDateTime	Returns an expression formatted as a date or time
Hour	Returns a number that represents the hour of the day (between 0 and 23, inclusive)
IsDate	Returns a Boolean value that indicates if the evaluated expression can be converted to a date
Minute	Returns a number that represents the minute of the hour (between 0 and 59, inclusive)
Month	Returns a number that represents the month of the year (between 1 and 12, inclusive)
MonthName	Returns the name of a specified month
Now	Returns the current system date and time
Second	Returns a number that represents the second of the minute (between 0 and 59, inclusive)
Time	Returns the current system time
Timer	Returns the number of seconds since 12:00 AM
TimeSerial	Returns the time for a specific hour, minute, and second
TimeValue	Returns a time
Weekday	Returns a number that represents the day of the week (between 1 and 7, inclusive)
WeekdayName	Returns the weekday name of a specified day of the week
Year	Returns a number that represents the year

Expression Functions

Expressions	Description
Eval	Evaluates an expression and returns the result
RegExp	Provides simple regular expression support.

Format Functions

Function	Description
FormatCurrency	Returns an expression formatted as a currency value
FormatDateTime	Returns an expression formatted as a date or time
FormatNumber	Returns an expression formatted as a number
FormatPercent	Returns an expression formatted as a percentage

I/O Functions

Input/Output	Description
InputBox	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.
MsgBox	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.
LoadPicture	Returns a picture object

Math Functions

Function	Description
Abs	Returns the absolute value of a specified number
Atn	Returns the arctangent of a specified number
Cos	Returns the cosine of a specified number (angle)
Exp	Returns *e* raised to a power
Hex	Returns the hexadecimal value of a specified number
Int	Returns the integer part of a specified number
Fix	Returns the integer part of a specified number
Log	Returns the natural logarithm of a specified number
Oct	Returns the octal value of a specified number
Randomize	Initializes the random-number generator
Rnd	Returns a random number less than 1 but greater or equal to 0
Sgn	Returns an integer that indicates the sign of a specified number
Sin	Returns the sine of a specified number (angle)
Sqr	Returns the square root of a specified number
Tan	Returns the tangent of a specified number (angle)

Miscellaneous Functions

Miscellaneous	Description
GetLocale	Returns the current locale ID
RGB	Returns a whole number representing an RGB color value
SetLocale	Sets the current locale ID

Script Engine Functions

Script Engine ID	Description
ScriptEngine	Returns a string representing the scripting language in use
ScriptEngineBuildVersion	Returns the build version number of the scripting engine in use
ScriptEngineMajorVersion	Returns the major version number of the scripting engine in use
ScriptEngineMinorVersion	Returns the minor version number of the scripting engine in use

String Functions

Function	Description
InStr	Returns the position of the first occurrence of one string within another. The search begins at the first character of the string
InStrRev	Returns the position of the first occurrence of one string within another. The search begins at the last character of the string
LCase	Converts a specified string to lowercase
Left	Returns a specified number of characters from the left side of a string
Len	Returns the number of characters in a string
LTrim	Removes spaces on the left side of a string
Mid	Returns a specified number of characters from a string
Replace	Replaces a specified part of a string with another string a specified number of times
Right	Returns a specified number of characters from the right side of a string
RTrim	Removes spaces on the right side of a string
Space	Returns a string that consists of a specified number of spaces
StrComp	Compares two strings and returns a value that represents the result of the comparison
String	Returns a string that contains a repeating character of a specified length
StrReverse	Reverses a string
Trim	Removes spaces on both the left and the right side of a string
UCase	Converts a specified string to uppercase

Variant Identification Functions

Variant Function	Description
IsArray	Returns a Boolean value indicating whether a variable is an array
IsDate	Returns a Boolean value indicating whether an expression can be converted to a date
IsEmpty	Returns a Boolean value indicating whether a variable has been initialized.
IsNull	Returns a Boolean value that indicates whether an expression contains no valid data (Null).
IsNumeric	Returns a Boolean value indicating whether an expression can be evaluated as a number
IsObject	Returns a Boolean value indicating whether an expression refers to a valid Automation object.
TypeName	Returns a string that provides Variant subtype information about a variable
VarType	Returns a value indicating the subtype of a variable

Statements

VBScript statements are used to perform fundamental operations such as decision making, repetition (looping) and assignments. Statements combined with Operators are the building blocks for more complex code.

Multiple statements can appear on the same line as long as they are separated by a colon (**:**). For purposes of code readability, it is recommended to use one statement per line.

Assignment Statements

Many of VBScripts assignment statements have already been covered. For consistency purposes, they are listed here. Please refer to the Appendix for a more detailed description of their use.

Assignment Statements

Statement	Description
Const	Declares constants for use in place of literal values
Dim	Declares variables and allocates storage space
Erase	Reinitializes the elements of fixed-size arrays, deallocates dynamic-array storage space.
Option Explicit	Forces explicit declaration of all variables in the script
Private	Declares private variables and allocates storage space
Public	Declares public variables and allocates storage space
ReDim	Declare dynamic array variables, allocates or reallocates storage space at procedural level

Comment Statements

Comment statements are used to provide documentation comments with the code.

Comment Statements

Comments	Description
Rem	Includes explanatory remarks in a program
‘	Includes explanatory remarks in a program (single quote)

Control Flow Statements

By default, VBScript sequentially moves (flows) through the script from statement to statement. As is typical with virtually all high-level programming languages, control flow statements can alter this flow by branching to other code sections based upon logic conditions, inputs, errors, etc.

One of the most commonly used control flow statement is the **If..Then..Else** statement. This control flow statement takes the following format:

{simple format} **If** condition **Then** statement(s) [**Else** elsestatement(s)]

{block format}

```
If condition Then
    [statement(s)]
[ElseIf condition-n Then
    [elseifstatement(s)]] . . .
[Else
    [elsestatement(s)]]
End If
```

The condition can be a boolean constant or boolean variable, or a numeric or string expression that evaluates to **True** or **False**.

Refer to the Appendix for a detail description of these functions.

Control Flow Statements

Function	Description
Do…Loop	Repeats a block of statements while a condition is True or until a condition becomes True
Execute	Executes one or more specified statements
Execute Global	Executes one or more specified statements in the global namespace of a script
Exit Do	Exit a Do Loop Function. Transfers control to the statement following the Loop statement.
Exit For	Exit a For Loop Function (For…Next or For Each…Next loop). Transfers control to the statement following the Next statement.
For...Next	Repeats a group of statements a specified number of times
For Each…Next	Repeats a group of statements for each element in an array or collection
If…Then…Else	Conditionally executes a group of statements, depending on the value of an expression
Select Case	Executes one of several groups of statements, depending on the value of an expression
While…Wend	Executes a series of statements as long as a given condition is **True**
With…End With	Executes a series of statements on a single object

Procedure Statements

There are two types of procedure statements; the **Sub** procedure and the **Function** procedure. Both of these procedure statements are intended to encapsulate a set of statements that provide functionality that can be repeatedly called, but the difference between the two is how arguments are passed and results returned.

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

The **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 in the **Function** name. 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**.

Procedure Statements

Function	Description
Call	Transfers control to a Sub or Function procedure
End Function	Immediately exits a Function procedure
End Sub	Immediately exits a Sub procedure
Exit Function	Exit a Function, generally as a result of a condition
Exit Sub	Exit a Subroutine, generally as a result of a condition
Function	Declares the name, arguments, and code that form the body of a Function procedure
GetRef	Associates an event handler with a specific function
Sub	Declares the name, arguments, and code that form the body of a Sub procedure (Subroutine).

Objects and Classes

Traditional programming is made up of a collection of subroutines and functions that are typically processed in a sequential or looping manner. In contrast, object oriented programming is a different programming methodology where a program is viewed as being composed of a collection of individual objects. These objects process data and can interact with other objects directly without having to be explicitly programmed to do so. The advantages claimed by object-oriented program include code reusability, rapid deployment of large-scale complex tasks, and ease of use/debugging. Today, object-oriented programming is widely used and is supported with both programming languages (e.g. VB.NET, C++, Visual C++) and operating systems (e.g. Microsoft's .NET architecture). Object-oriented programming has also become popular within scripting languages, such as VBScript. Beginning with VBScript 5.0, developers have been able to use user-defined Classes.

The key concepts with object-oriented programming include:

- **Class**
  The class is the highest level that defines a unit (set) of data and its behavior. Classes form the basis for modularity and structure in an object-oriented program. The class should sufficiently describe the set of data, and the code for a class should be contained within it and be self-sufficient (except for operating system support). While the terms classes and objects often get used interchangeably, classes describe the structure of objects. One way to think of a class is that it is a container for code. It can also be viewed as a template for an object. When a class is declared (instantiated) by the **Set** statement, it then becomes an object and memory is allocated for it.

- **Object**
  An object is an in-memory instance of a class. In computer science terms, it is a run-time manifestation (instantiation) of a particular exemplar of a class. Each object has its own data, but the code within a class can be shared (for efficiency). Programs generally have multiple objects. Multiple copies (objects) of a given class can be created. Objects are temporary, i.e. they can be created and removed at will, depending on the programming needs.

- **Encapsulation**
  Encapsulation wraps the data and functions into a single unit, ensuring that the object can be changed only through established interfaces. Encapsulation is sometimes referred to as information hiding. Some of these common interfaces are:
  - Fields
    Fields are simply public variables stored within the object, as defined by the class. These variables store items of information about an object.
  - Properties
    Properties, like fields, also store items of information on an object. But Properties use Property procedures to control how values are set or returned. VBScript has two primary Property procedures; **Let** and **Get**. The **Get** property procedure retrieves a Property value, while the **Let** Property procedure assigns a value to the property value. A third Property procedure **Set** is used with an Object inside of the Class block.
  - Methods
    Methods are a collection of subroutines (**Sub**) and function procedures (**Function**) declared within a class.
  - Events
    An event is a message sent by an object announcing that something important has happened.

  Access of an object's methods, properties and fields are made by referring to the object, followed by a period, then the particular method, property or field of interest. E.g.

  ```
  Object.Method
  Object.Property
  ```

Object.Property.Item

- **Dynamism**
  Dynamism relates to the method of allocating computer resources and definition resources required to run an object-oriented program. There are different types, but VBScript used late-bound (late-binding) dynamic typing. This means that the VBScript engine will make the object type determination at runtime and allocate sufficient memory at that time. Note that VBScript and VB.NET are slightly different in their approach to dynamism, and therefore they can declare some variables and objects in different manners (although many forms of declaration are the same).
- **Outlet Connections**
  At times, Objects will connect together and this connection needs to be defined. With IWS, an example of a connection would be between a VBScript object (e.g. ADODB) and a Database Provider (a Provider is a front-end to a database). This connection needs to be defined, and then the connection string (of parameters) between the objects gets defined. When the need for the connection is finished, the connection should be closed.

While a full treatment of object-oriented programming is beyond the scope of these materials, the fundamental concepts of Objects and Classes are important to understand. VBScript supports COM-based Objects (Component Object Module, a Microsoft standard) such as the ActiveX controls, ADO.NET, FileSystemObject, and Microsoft Office Automation objects. VBScript also supports user-defined classes, or Class Objects.

VBScript COM objects and VBScript Class objects differ from each other in several important respects. These differences lead to each type of object having its unique strengths:

- VBScript classes are more flexible than VBScript COM objects. Class Objects have an abstract subtype that encapsulates the data you want and the functions you need to work with that data. VBScript COM objects have only basic subtypes (integer or string).
- VBScript classes are slightly more efficient than COM objects. The VBScript parser can execute the classes' code directly instead of asking the COM object to execute a method.
- COM objects are binary modules. VBScript classes are ASCII files.
- You can use any scripting language to write COM objects. You can only use VBScript to write VBScript classes.
- You can use COM objects from within any development environment that supports COM automation. VBScript classes can only be used within development and runtime environments that support VBScript (e.g IWS and Microsoft Internet Explorer).

VBScript Object Commands

VBScript includes several Functions and Statements that can be used to access objects, including their methods and properties. There are a large variety of objects available to VBSript, including user-defined objects, intrinsic objects and extrinsic objects.

VBScript Object Functions

Function	Description
CreateObject	Creates and returns a reference to an Automation object
GetObject	Returns a reference to an Automation object from a file
IsObject	Returns a Boolean value indicating whether an expression references a valid Automation object.

Object Statements

Statement	Description
Class	Declares the name of a class, as well as a definition of the variables, properties, and methods that comprise the class
Exit Property	Forces an exit from inside a Property Set function.
For Each…Next	Repeats a group of statements for each element in an array or a collection.
Property Get	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
Property Let	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
Property Set	Sets a reference to an object
Set	Assigns an object reference to a variable or property, or associates a procedure reference with an event. Usually used to instantiate an object.

Error Handling Statements

Statement	Description
On Error	Enables or disables error-handling

Object & Collection Summary

Objects & Collections	Description
Debug	The Debug object is an intrinsic global object that can send an output to a script debugger, such as the Microsoft Script Debugger.
Dictionary	An associative array that can store any type of data. Data is accessed by a key.
Drive	An object that refers to a specific Drive
Drives	A collection of Drive objects.
Err	Contains information about the last run-time error. Accepts the Raise and Clear methods for generating and clearing run-time errors.
File	An object that refers to a specific File
Files	A collection of File objects.
FileSystemObject	An object model used to access the Windows file system
Folder	An object that refers to a specific Folder
Folders	A collection of Folder objects.
Match	Provides access to the read-only properties of a regular expression match.
Matches	Collection of regular expression Match objects.
RegExp	Provides simple regular expression support.
Submatches	A collection of regular expression submatch strings.
TextStream	An object that refers to a text File

VBScript User-Defined Class Objects

To define a user-defined Class Object, you use the **Class** statement to declare a class. The **End Class** statement defines the termination of the Class. Together, these statements form a Class construct, or Class block. E.g.

```
Class objName
    ' Place the Class variables, Properties and Methods here
End Class
```

In this syntax, *objName* is the name given to the Class Object. The class object name must follow standard VBScript variable naming conventions. Class Objects are usually declared in the variable definition sections. You can have multiple Class blocks in a single VBScript file, but each block must contain the **Class …End Class** statements. Classes cannot be nested.

Once you have defined the Class Object, you need to create an instance of the Class, similar to how other objects are created. When the Class Object is instantiated, memory is allocated for the Class Object. The **Set** statement is used with the **New** keyword to assign an instance of the class to a variable. With VBScript, this is the only time the **New** keyword is used (i.e. to instantiate a user-defined Class). E.g.

```
Dim MyObj
Set MyObj  = New objName
```

The Object name *MyObj* is the Object variable name, and must follow standard VBScript variable naming conventions. The Object variable name is a reference (address) of the Object stored in memory, it is not the Object itself.

Inside the Class block, any Class variables, Properties, Methods and Events can be defined by the developer. The developer does not have to use all of the capabilities of the Class construct, i.e. Classes can be created without Methods or Properties. The design of the Class Object is completely up to the developer.

Class variables are created within the Class structure by using the **Dim**, **Public**, or **Private** statements. Variables defined within the Class structure by any of these statements must follow the standard VBScript variable naming conventions. Variables can be simple variables or arrays. E.g.

```
Class className
    Dim  var1, var2
    Public var3, var4
    Private var5, var6
End Class
```

The choice of the **Dim**, **Public**, or **Private** statements determine whether the variable is accessible outside of the Class Object. Variables are public by default, i.e. they are accessible outside of the Class Object. Both the **Dim** and the **Public** statements create public variables, while the **Private** statement creates variables that are not public. As a general rule, it is good programming practice to make all Class variables private, since the developer will want to tightly control when these variables are changed.

VBScript does not support Class-level Constants, i.e. named constants declared at the Class level. You cannot use the **Const** statement at the Class-level so that a constant can be used throughout a Class, but you can use the **Const** statement within a Property or Method. However, the constant will only have local scope within the Property or Method.

Class Object variables are accessible to VBScript code outside the Class through Class Properties. Class Properties "wrap" the Private variables of a Class. Inside the Class block, the Properties are defined by **Property Get [|Let|Set] … End Property** statement(s). For VBScript code outside the Class, the Property is accessed by referencing the Object Name.Property.

There are different type of Class Properties, depending on whether the Class variable is to be read, written to, or the Class variable is itself a Class Object. These Properties can be declared Public or Private.

Property Get

The Property Get procedure is used to access (return) private variables inside of the Class structure that are used as a read-only Property, or the read portion of a read-write Property. For VBScript code outside the Class, this type of Class Object Property is generally assigned to a variable or used in a conditional expression. The Property Get procedure returns a value to the calling code, and is general not used with any arguments. [Note: VBScript will let you add arguments to the Property Get procedure, but if you do so you must add the additional argument to the corresponding Property Let or Property Set procedure, since Property Let/Property Set must have one more argument than the corresponding Property Get procedure. It is generally considered bad programming form to have arguments in the Property Get procedure].

Property Let

The Property Let procedure is used to access (assign) private variables inside of the Class structure that are used as a write-only Property or are the write portion of a read-write Property. For VBScript code outside of the Class, this type of Class Object Property is usually assigned by a variable or a constant.

Property Set

The Property Set procedure is exclusively used when the Class Object needs to store Properties that are object-based instead of numeric, date, boolean or string subtype variables. Property Set replaces the Property Let procedure. While Property Set and Property Let are functionally similar, there are two key differences:

1. With the **Property Set** procedure, in the VBScript code segment (outside the Class block) you must use the syntax

   ```
   Set Object1.Property = Object2
   ```

   This is because VBScript does not let you use the assignment operator (=) to assign objects without the **Set** command.

2. The **Property Set** procedure makes it clear that the Property is an object-based Property

Example:

```
Class FileSpec                              ‘ Define a Class block
Private master_file
Private master_FSO
Public Property Let FileName(strName)  ‘ Define a Public Property to assign the file name
    master_file = strName
End Property
Public Property Get FileName            ‘ Define a Public Property to retrieve a file name
    FileName = master_file
End Property
Public Property Set FSO(m_FSO)              ‘ Define a Public Property for an object
    Set master_FSO = m_FSO
End Property
End Class

Rem Below is the VBScript code

Dim objFSO                                  ‘ Declare variables and objects
Dim objFilePointer, cur_file
Set objFSO = CreateObject(“Scripting.FileSystemObject”)          ‘ Instantiate the COM object
Set objFilePointer = New FileSpec           ‘ Instantiate the Class Object
objFilePointer.FileName = “Myfile.mdb”  ‘ Assigns “Myfile.mdb” as the file name
```

```
        cur_file = objFilePointer.FileName                    ' Retrieves the current file name "Myfile.MDB"
        Set objFilePointer.FSO = objFSO                       ' Assigns an Object to the Property
        Set objFilePointer = Nothing                          ' Keyword Nothing releases the object memory
```

A couple notes on the example above. The **CreateObject** command is used to instantiate an Object that is known at the system level (e.g. a COM object). Also, so far this example only shows how to assign and retrieve property values. It is generally the Method(s) that control the action an object performs, not the properties.

A Property can be made read-only by only providing a Property Get procedure, or by declaring the Property Let procedure as Private instead of Public. A Property can be made write-only by only providing the Property Let procedure, or by declaring the Property Get procedure as Private instead of Public.

Class Methods are really just **Functions** and **Subroutines** inside of a Class block. These functions and subroutines can be either Private or Public. If they are public, they will be accessible to a VBScript code segment outside of the Class block by referencing the obj.Method. If they are private, they will only be available to code within the Class block.

An example of Class Methods is as follows:

```
    Class FileSpec
        Private master_file
        Private master_FSO      Private master_file
        Private Sub Class_Initialize                          ' Class Object initialization code
            ' code goes here
        End Sub
        Private Sub Class_Terminate                           ' Class Object termination code
            ' code goes here
        End Sub
        Public Property Let FileName(strName)  ' Define a Public Property to assign the file name
            master_file = strName
        End Property
        Public Property Get FileName           ' Define a Public Property to retrieve a file name
            FileName = master_file
        End Property
        Public Property Set FSO(m_FSO)                        ' Define a Public Property for an object
            Set master_FSO = m_FSO
        End Property
        Public Sub Delete                                     'Method to delete the master file
            master_FSO.DeleteFile (master_file)
        End Sub
    End Class

    Rem Below is the VBScript code
        Dim objFSO                                            ' Declare variables and objects
        Dim objFilePointer, cur_file
        Set objFSO = CreateObject("Scripting.FileSystemObject")            ' Instantiate the COM object
        Set objFilePointer = New FileSpec                     ' Instantiate the Class Object
        objFilePointer.FileName = "Myfile.mdb"  ' Assigns "Myfile.mdb" as the file name
        cur_file = objFilePointer.FileName                    ' Retrieves the current file name "Myfile.MDB"
        Set objFilePointer.FSO = objFSO                       ' Assigns an Object to the Property
        objFilePointer.Delete                                 ' Executes a Method to delete a file
        Set objFilePointer = Nothing                          ' Keyword Nothing releases the object memory
```

VBScript Class Objects automatically supports two type of Class Events; **Class_Initialize** and **Class_Terminate** Events. The code inside the Class_Initialize event executes once when an Object based on the Class is first instantiated. Any code put in this event is optional, and is typically used for initialization. Code inside the Class_Terminate event executes once just before the Object based on the Class is destroyed (i.e. Set to Nothing, or the Object goes out of scope). Usage is as follows:

```
    Class FileSpec
    Private master_file
    Private master_FSO      Private master_file
    Private Sub Class_Initialize                        ‘ Class Object initialization code
        ‘ code goes here
    End Sub
    Private Sub Class_Terminate                         ‘ Class Object termination code
        ‘ code goes here
    End Sub
    Public Property Let FileName(strName)  ‘ Define a Public Property to assign the file name
        master_file = strName
    End Property
    Public Property Get FileName               ‘ Define a Public Property to retrieve a file name
        FileName = master_file
    End Property
    Public Property Set FSO(m_FSO)                      ‘ Define a Public Property for an object
        Set master_FSO = m_FSO
    End Property
End Class
```

VBScript Objects and Collections

VBScript has certain Objects and Collections that are inherent with VBScript. These include:

- Debug
- Err Object
- Match Object & Matches Collections
- Scripting Dictionary Object
- Scripting FileSystemObject
    - Drive Object
    - File Object
    - FileSystemObject Collections
    - Folder Object
- Regular Expression Object & Submatches Collection
- TextStream Object

VBScript Implicit Objects and Collections

Objects & Collections	Description
Class Object	Declares the name of a class, as well as a definition of the variables, properties, and methods that comprise the class
Debug	The Debug object is an intrinsic global object that can send an output to a script debugger, such as the Microsoft Script Debugger.
Err	Contains information about the last run-time error. Accepts the Raise and Clear methods for generating and clearing run-time errors.
Match Object	
Dictionary	An associative array that can store any type of data. Data is accessed by a key.
Matches Collection	
RegExp Object	
SubMatches Collection	

Object & Collection Summary

Objects & Collections	Description
Drive	An object that refers to a specific Drive
Drives	A collection of Drive objects.
File	An object that refers to a specific File
Files	A collection of File objects.
FileSystemObject	An object model used to access the Windows file system
Folder	An object that refers to a specific Folder
Folders	A collection of Folder objects.
Match	Provides access to the read-only properties of a regular expression match.
Matches	Collection of regular expression Match objects.
RegExp	Provides simple regular expression support.
Submatches	A collection of regular expression submatch strings.
TextStream	An object that refers to a text File

Err Object

The VBScript **Err** object contains information about run-time errors.

Err Object Properties

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

Err Object Methods

Properties	Description
Clear	Clears all property settings.
Raise	Generates a run-time error.

The properties of the **Err** object are set by the generator of an error-Visual Basic, an Automation object, or the VBScript programmer.

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

When a run-time error occurs, the properties of the **Err** object are filled with information that uniquely identifies the error and information that can be used to handle it. To generate a run-time error in your code, use the VBScript Err Object Raise Method. The Err object's properties are reset to zero or zero-length strings ("") after an **On Error Resume Next** statement. The VBScript Err Object Clear Method can be used to explicitly reset **Err**.

The **Err** object is an intrinsic object with global scope-there is no need to create an instance of it in your code.

Scripting Dictionary Object

A dictionary object is part of the Scripting type library. The dictionary object is a special type of an array which stores a data item that is associated with a unique key. The key, which is usually a number or a string, is used to retrieve an individual item. You can use a Dictionary when you need to access random elements frequently or need to access information contained in the array based on its value, not position.

The Dictionary object has both Methods and Properties that can be used to manipulate the Dictionary.

Dictionary Methods

Method	Description
Add	Adds a key and item pair
Exists	Indicates if a specific key exists
Items	Returns an array containing all items in a Dictionary object
Keys	Returns an array containing all keys in a Dictionary object
Remove	Removes a key, item pair
RemoveAll	Removes all key, item pairs

Dictionary Properties

Method	Description
CompareMode	The comparison mode for string keys
Count	The number of items in a Dictionary object
Item	An item for a key
Key	A key

The following code creates a Dictionary object and adds items and keys:

```
Dim d 'Create a variable
Set d = CreateObject("Scripting.Dictionary")
d.Add "a", "Athens"                               'Add some keys and items
d.Add "b", "Belgrade"
d.Add "c", "Cairo"
```

Scripting FileSystemObject

The VBScript FileSystemObject object provides access to a computer's file system

FileSystemObject Methods

Method	Description
BuildPath	Appends a name to an existing path.
CopyFile	Copies one or more files from one location to another.
CopyFolder	Recursively copies a folder from one location to another.
CreateFolder	Creates a folder.
CreateTextFile	Creates a specified file name and returns a TextStream object.
DeleteFile	Deletes a folder and its contents.
DeleteFolder	Deletes a folder and its contents.
DriveExists	Indicates the existence of a drive.
FileExists	Indicates the existence of a file.
FolderExists	Indicates the existence of a folder.
GetAbsolutePathName	Returns a complete and unambiguous path from a provided path specification.
GetBaseName	Returns the base name of a path.
GetDrive	Returns a Drive object corresponding to the drive in a path
GetDriveName	Returns a string containing the name of the drive for a path.
GetExtensionName	Returns a string containing the extension for the last component in a path.
GetFile	Returns a File object corresponding to the file in a path.
GetFileName	Returns the last component of a path that is not part of the drive specification.
GetFolder	Returns a Folder object corresponding to the folder in a specified path.
GetParentFolderName	Returns a string containing the name of the parent folder of the last component in a path.
GetSpecialFolder	Returns the special folder requested.
GetTempName	Returns a randomly generated temporary file or folder name.
MoveFile	Moves one or more files from one location to another.
MoveFolder	Moves one or more folders from one location to another.
OpenTextFile	Opens a file and returns a **TextStream** object

FileSystemObject Properties

Properties	Description
Drives	A **Drives** collection of all **Drive** objects available on the local machine.

Collections returned by **FileSystemObject** method calls reflect the state of the file system when the collection was created. Changes to the file system after creation are not reflected in the collection. If the file system might be changed during the lifetime of the collection object, the method returning the collection should be called again to ensure that the contents are current.

```
Set fs = CreateObject("Scripting.FileSystemObject")
Set a = fs.CreateTextFile("c:\testfile.txt", True)
a.WriteLine("This is a test.")
a.Close
```

In the code shown above, the CreateObject function returns the FileSystemObject (fs). The CreateTextFile method then creates the file as a TextStream object (a) and the VBScript TextStream Object WriteLine Method writes a line of text to the created text file. The VBScript TextStream Object Close Method flushes the buffer and closes the file.

Drive Object
The **Drive** object provides access to the properties of a particular disk drive or network shared drive.

Drive Object Properties

Properties	Description
AvailableSpace	The amount of space available to a user on the specified drive or network share.
DriveLetter	The drive letter of a physical local drive or network share
DriveType	A value indicating the type of a drive.
FileSystem	The amount of free space available to a user on the drive or network share.
FreeSpace	The amount of free space available to a user on the drive or network share.
IsReady	True if the drive is ready, False if not.
Path	The file system path for a drive.
RootFolder	A Folder object representing the root folder of a drive.
SerialNumber	The decimal serial number used to uniquely identify the disk volume.
ShareName	The network share name of a drive
TotalSize	The total space, in bytes, of a drive or network share
VolumeName	The volume name of a drive.

The following code illustrates the use of the Drive object to access drive properties:

```
Sub ShowFreeSpace(drvPath)
Dim fs, d, s
Set fs = CreateObject("Scripting.FileSystemObject")
Set d = fs.GetDrive(fs.GetDriveName(drvPath))
s = "Drive " & UCase(drvPath) & " - "
s = s & d.VolumeName & vbCrLf
s = s & "Free Space: " & FormatNumber(d.FreeSpace/1024, 0)
s = s & " Kbytes"
Response.Write s
End Sub
```

File Object
The **File** object provides access to all the properties of a file.

File Object Methods

Properties	Description
Copy	Copies a file from one location to another.
Delete	Deletes a file.
Move	Moves a file from one location to another.
OpenAsTextStream	Opens a file and returns a **TextStream** object.

File Object Properties

Properties	Description
Attributes	The attributes of a file.
DateCreated	The date and time that the file was created.
DateLastAccessed	The date and time that the file was last accessed.
DateLastModified	The date and time that the file was last modified.
Drive	The drive letter of the drive on which the file resides.
Name	The name of the file.
ParentFolder	The Folder object for the parent of the file.
Path	The file system path to the file.
ShortName	The short name used by programs that require 8.3 names.
ShortPath	The short path use by programs that require 8.3 names.
Size	The size, in bytes, of a file.
Type	Information about the type of a file.

The following code illustrates how to obtain a File object and how to view one of its properties.

```
Sub ShowFileInfo(filespec)
Dim fs, f, s
Set fs = CreateObject("Scripting.FileSystemObject")
Set f = fs.GetFile(filespec)
s = f.DateCreated
Response.Write s
End Sub
```

Folder Object

The VBScript **Folder** object provides access to all the properties of a folder.

Folder Object Methods

Properties	Description
Copy	Copies a folder from one location to another.
Delete	Deletes a folder.
Move	Moves a folder from one location to another.
CreatTextFile	Creates a file and returns a **TextStream** object.

Folder Object Properties

Properties	Description
Attributes	The attributes of a folder.
DateCreated	The date and time a folder was created.
DateLastAccessed	The date and time that the folder was last accessed.
DateLastModified	The date and time that the folder was last modified.
Drive	The drive letter of the drive on which the folder resides.
Files	A **Files** collection of all **File** objects in the folder.
IsRootFolder	**True** if this is the root folder of a drive.
Name	The name of the folder.
ParentFolder	The **Folder** object for the parent of the folder.
Path	The file system path to the folder.
ShortName	The short name used by programs that require 8.3 names.
ShortPath	The short path used by programs that require 8.3 names.
Size	The size, in bytes, of all files and subfolders contained in a folder
SubFolders	A **Folders** collection containing all the folders in a **Folder** object

The following code illustrates how to obtain a **Folder** object and how to return one of its properties:

```
Sub ShowFolderInfo(folderspec)
Dim fs, f, s,
Set fs = CreateObject("Scripting.FileSystemObject")
Set f = fs.GetFolder(folderspec)
s = f.DateCreated
Response.Write s
End Sub
```

Example

```
Const OverWrite = TRUE
Const DeleteRdOnly = True

SourceFile = “C:\data\MyData.MDB”
SourceFiles = “C:\data\*.MDB”
DestPath = “C:\Backup”
DeleteFile = “C:\backup\Mydata.MDB”
DeleteFiles = “C:\backup\*.MDB)

Set objFS = CreateObject(“Scripting.FileSystemObject”)

‘  Copy a single file to a new folder, overwrite any existing file in destination folder
objFS.CopyFile (SourceFile, DestPath, OverWrite)

‘ Copy a set of files to a new folder, overwrite any existing files in destination folder
objFS.CopyFile (SourceFiles, DestPath. OverWrite)

‘ Delete a file
objFS.DeleteFile(DeleteFile)

‘ Delete a set of files in a folder
objFS.DeleteFile(DeleteFiles, DeleteRdOnly)

‘ Move a file to a new folder
objFS.MoveFile(SourceFile, DestPath)

‘ Move a set of files to a new folder
objFS.MoveFile(SourceFiles, DestPath)

‘ Rename a file
objFS.MoveFile(SourceFile, “C:\data\MyData041406.MDB”)

‘ Verify if a file exists
If objFS.FileExists (SourceFile) Then
    Set objFolder =objFS.GetFile(SourceFile)
    MsgBox “File Exists “ & objFolder              ‘ Will display “File Exists “ and Path + File
Else
    MsgBox “File does not exist”
End If
```

VBScript Drives Collection

Read-only collection of all available drives. Removable-media drives need not have media inserted for them to appear in the Drives collection.

Drives Collection Object Properties

Properties	Description
Count	Returns the number of items in a collection. Read-only
Item	Returns an item on the specified key. Read/Write

The following code illustrates how to get the Drives collection and iterate the collection using the For Each...Next statement:

```
Sub ShowDriveList
Dim fs, d, dc, s, n
Set fs = CreateObject("Scripting.FileSystemObject")
Set dc = fs.Drives
For Each d in dc
  s = s & d.DriveLetter & " - "
If d.DriveType = Remote Then
  n = d.ShareName
Else
  n = d.VolumeName
End If
s = s & n & vbCrLf
Next
Response.Write s
End Sub
```

VBScript Files Collection

Collection of all **File** objects within a folder.

Files Collection Object Properties

Properties	Description
Count	Returns the number of items in a collection. Read-only
Item	Returns an item on the specified key. Read/Write

The following code illustrates how to get a Files collection and iterate the collection using the For Each...Next statement:

```
Sub ShowFolderList(folderspec)
Dim fs, f, f1, fc, s
Set fs = CreateObject("Scripting.FileSystemObject")
Set f = fs.GetFolder(folderspec)
Set fc = f.Files
For Each f1 in fc
s = s & f1.name
s = s & vbCrLf
Next
Response.Write s
End Sub
```

VBScript Folders Collection
Collection of all Folder objects contained within a Folder object.

Folders Collection Methods

Properties	Description
Add	Adds a new **Folder** to a **Folders** collection

Folders Collection Properties

Properties	Description
Count	Returns the number of items in a collection. Read-only
Item	Returns an item on the specified key. Read/Write

The following code illustrates how to get a Folders collection and how to iterate the collection using the For Each...Next statement:

```
Sub ShowFolderList(folderspec)
Dim fs, f, f1, fc, s
Set fs = CreateObject("Scripting.FileSystemObject")
Set f = fs.GetFolder(folderspec)
Set fc = f.SubFolders
For Each f1 in fc
s = s & f1.name
s = s & vbCrLf
Next
Response.Write s
End Sub
```

TextStream Object
The VBScript **TextStream** object facilitates sequential access to a file

TextStream Object Methods

Properties	Description
Close	Closes an open stream.
Read	Reads a specified number of characters from a stream.
ReadAll	Reads an entire stream.
ReadLine	Reads an entire line from a stream.
Skip	Skips a specified number of characters when reading a stream.
SkipLine	Skips the next line when reading a stream.
Write	Writes a specified string to a stream.
WriteBlankLines	Writes a specified number of newline characters to a stream.
WriteLine	Writes a specified string and newline character to a stream.

TextStream Object Properties

Properties	Description
AtEndOfLine	**True** if the file pointer is before the end-of-line marker.
AtEndOfStream	**True** if the file pointer is at the end of the stream
Column	The column number of the current character in the stream.
Line	The current line number of the stream.

VBScript TextStream Object

Description:	The VBScript TextStream object
Usage:	oTextStream.{property | method}
Return:	Depends on Property or Method used
Remarks	
Example:	In the following code, a is the TextStream object returned by the CreateTextFile method on the FileSystemObject:

```
Set fs = CreateObject("Scripting.FileSystemObject")
Set a = fs.CreateTextFile("c:\testfile.txt", True)
a.WriteLine("This is a test.")
a.close
```

COM Objects and Collections

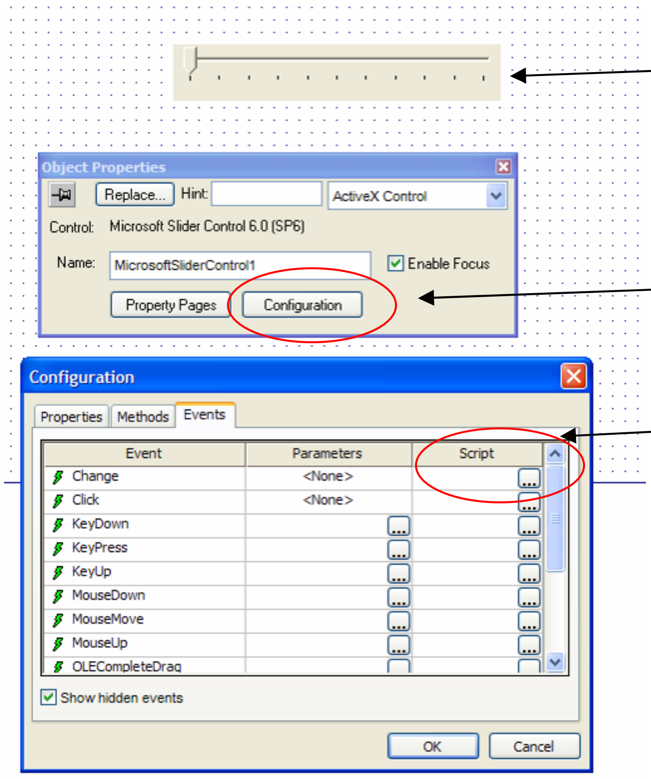
In addition to user-defined Class Objects and VBScript Objects and Collections, there are many different COM Objects (and Object Collections) and other system objects based on COM technology that are accessible from VBScript. These Objects include:

- ActiveX Controls inserted on an IWS Screen (via Insert OCX tool)
- ActiveX Controls instantiated via VBScript
- ADODB and ADOX Objects and Collections
- Microsoft Office OLE Automation (Word, Excel, Access, Outlook & Components)
- WMI
- WSH
- WSDL
- XMLDOM

ActiveX Controls Inserted On An IWS Screen

InduSoft Web Studio (IWS) serves as an ActiveX control container, which is a parent program that supplies the environment for an ActiveX control to run. Through the IWS development interface (insert OCX tool), one or more ActiveX controls can be added to a screen. The OCX (ActiveX Control) must first be registered, if it was not already done so as part of the installation of the ActiveX control. IWS provides a Register Controls tool (under Tools on the toolbar) to allow registration of ActiveX controls, and to verify if a control has already been registered.

After the OCX is inserted on the screen, IWS will assign the control a name. This name can be changed in the Object Properties dialog box, accessed by double clicking on the control in the IWS development environment, but the name of the control must be unique from any other control used by the current IWS application. In the Object Properties dialog box, the Configuration button will provide access to the Properties, Methods and Events accessible for this ActiveX control. In the Configuration dialog box, there is a tab for Events, which allow for the execution of a VBScript code segment if an Event is triggered for the ActiveX control. In the Properties and Methods tabs, parameters, triggers, IWS tags, etc. can be tied to the various Properties and Methods.


Microsoft Slider Control 6.0
Configure the Control's Properties, Methods & Events
Select to input VBScript code segments for the ActiveX Control Events
Object Properties
Replace...
Hint:
ActiveX Control
Property Pages
Configuration
Configuration
Properties
Methods
Events
Show hidden events
OK
Cancel

Control: Microsoft Slider Control 6.0 (SP6)

Name: MicrosoftSliderControl1 ☑ Enable Focus

Event	Parameters	Script
Change	<None>	...
Click	<None>	...
KeyDown	...	...
KeyPress	...	...
KeyUp	...	...
MouseDown	...	...
MouseMove	...	...
MouseUp	...	...
OLECompleteDrag		

Interaction with the ActiveX control from VBScript is accomplished through VBScript code placed in a Screen Script that is associated with the screen where the ActiveX control is placed. By entering a right mouse click on a blank portion of the screen, and selecting Screen Script, the Screen Script is accessed. For ActiveX Objects placed on the screen, you do not need to instantiate the Object in VBScript, IWS has already taken care of this. You simply need to reference the ActiveX control by its name, found in the Object Properties dialog box. **Note: when referring to the name from VBScript, the ActiveX control name is case sensitive**. From the VBScript screen interface, you can access the ActiveX control's Properties and Methods. Events are not accessible from the VBScript Screen Script interface. The Active

Key Notes:

- **You must use the VBScript Screen Script interface for the screen which contains the ActiveX control in order to access the ActiveX control's Properties and Methods. You cannot access the ActiveX control's Properties and Methods from another Screen Script, or from any other VBScript interface in IWS.**
- **From VBScript, you can only access the ActiveX control's Properties and Methods. VBScript code segments for Events that are triggered by the ActiveX control can be entered, but these VBScript code segments must be entered from the Configuration dialog box (i.e. Object Properties → Configuration → Events).**
- **When the ActiveX control is referenced from the VBScript Screen Script interface, the ActiveX control's name is case-sensitive.**
- **You do not need to instantiate the ActiveX control. IWS has already taken care of this. Simply refer to the ActiveX control name followed by a "." and then the Property or Method.**
- **In the VBScript Screen Script interface, place the cursor in a code segment area (Subroutine) and press Ctrl –Space to invoke IntelliSense to see the VBScript statements and functions, as well as the ActiveX controls available for this Script Interface.**
- **Once you enter the ActiveX control object name, when you type a period ("."), Intellisense will display a list of available Properties and Methods for the ActiveX control referenced.**

Additional information on this topic is covered in the *VBScript Configuration and Operation in IWS* section later in this material.

ActiveX Controls Instantiated from VBScript

ActiveX controls can be instantiated from VBScript by using the CreateObject and referencing the Program ID (ProgID) of the ActiveX object, although the ActiveX object will not show up on the IWS screen if the script segment is associated with a Screen.

ADODB and ADOX Objects and Collections

ADODB is the database wrapper for ADO.NET, or ActiveX Data Objects for Microsoft's .NET Framework. ADO.NET is Microsoft's database interface technology that provides an API to database client applications (i.e. IWS and VBScript), supporting a common interface to access and manipulate data contained in a wide variety of database servers from different vendors. From the database client side, there is a level of abstraction provided by the API that enables interaction (e.g. database access and manipulation) to various vendor's databases with virtually no code changes, except for the connection string to the database Provider (an object that interacts with the physical database). There are various ADODB Objects and Collections available to the developer.

ADOX are Microsoft's ActiveX Data Object Extensions for Data Definition Language (database schema creation, modification and deletion) and Security. It is a companion set of Objects to the core ADO.NET objects.

Microsoft Office Automation
VBScript can access the various Microsoft Office Automation COM servers. These include:

- Microsoft Access ("Access.Application")
- Microsoft Excel ("Excel.Application")
- Microsoft Word ("Word.Application")
- Microsoft Outlook ("Outlook.Application")
- Microsoft Graph
- Microsoft Excel Chart ("Excel.Chart")

To instantiate a Excel and a Word Application, for example, we would use the following VBScript statements:

```
Set objXL = CreateObject("Excel.Application")
Set objWrd = CreateObject("Word.Application")
```

Once the Microsoft Office COM object is instantiated, the VBScript Programmer can access the various Properties and Methods. Using VBScript, objects can be moved from one Microsoft application to another.

WMI
Windows Management Instrumentation, or WMI, is a set of extensions to the Windows Driver Model that provide an interface from a program (such as VBScript) into various components of the Windows operating system to retrieve information and notification. Using WMI and VBScript, management of Windows-based PCs and Servers can be accomplished either locally or remotely. WMI is based on the Common Information Model (CIM), allowing a uniform methodology of managing various Windows components. WMI is available to all .NET applications and is supported under Windows 2000, XP or Server 2003, but not Windows CE at present. Examples of Microsoft Windows components accessible through WMI include:

- Control Panel (can manipulate basic system settings and controls)
- Device Manager (display and control hardware added to the PC, which drivers are used)
- Event Viewer (view the system event log locally or remotely)
- RegEdit (Windows Registry Editor)
- Various applications (Notepad, Command.Com and Cmd.exe)
- Windows Core Components

Windows Script Host
Windows Script Host, or WSH, is the successor to the Batch File first introduced for DOS. WSH automates system administration tasks, and supports multi-lingual scripting including VBScript. Scripts can be run locally, or on remote computers. There are several WSH objects including:

- WScript Object (not available from IWS, since IWS is the host)
- WshShell (allows scripts to work with the Windows Shell – e.g. read/write to registry, shortcuts, system administration tasks, running programs)
- WshNetwork (manages network drives and printers)
- WshController (runs scripts locally or remotely)

WSDL
Web Services Definition Language, or WSDL, is an XML-based language for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. WSDL is frequently used in conjunction with SOAP (Simple Object Access Protocol, a simple XML-based protocol for applications to exchange information using HTTP). Common examples of WSDL are stock price, news services, weather information, currency conversion, etc.

VBScript code in an IWS application can instantiate a SOAP client object through the following statement:

```
Set oSOAP = CreateObject("MSSOAP.SoapClient")
```

XMLDOM
XMLDOM is the XML Document Object Model that defines a standardized approach for creating, accessing and manipulation XML documents. The DOM structures the XML document as a tree-like structure (the node), with each node having elements, attributes and text. There is a root element, which is the highest level element, and 0 or more child (sibling) nodes. Each node can also have 0 or more child nodes.

A VBScript code segment can be created in an IWS application to allow creation, accessing and manipulation of XML Documents. This allows passing of data between IWS and another computer in XML format. Note that in addition to XMLDOM, ADO.NET also supports XML databases.

VBScript Configuration and Operation in IWS

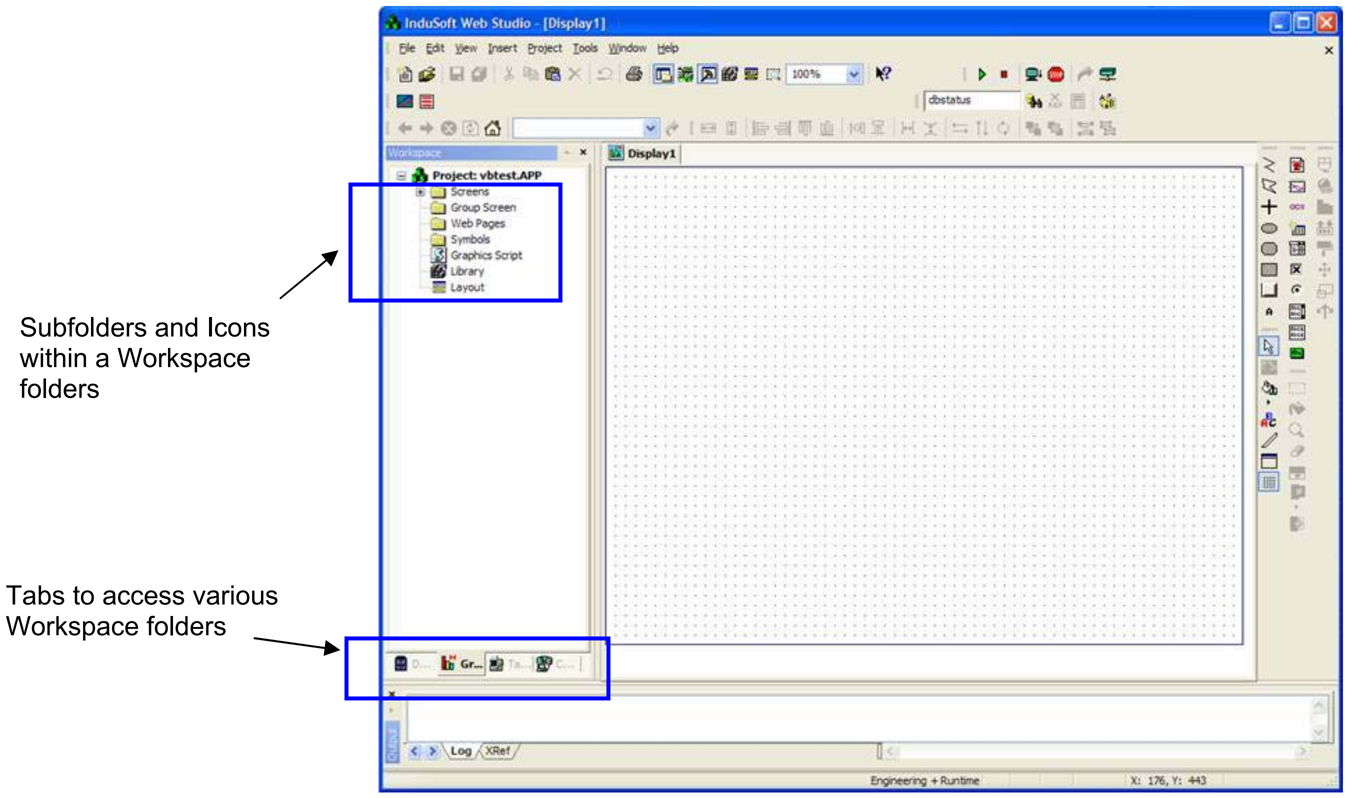
IWS acts as the host application for the Microsoft VBScript Engine. This means that to write VBScript, you need to be in the IWS development (engineering) environment. It is important to note that there is no one central location where a VBScript interface is located inside an IWS application. The location of the VBScript interface (where the VBScript code gets placed) depends on the function the VBScript code is to perform and the scope of access to its Procedures and Variables. InduSoft has implement VBScript in this manner to simplify its use, and to be consistent with the IWS architecture as well as current licensing methods.

VBScript is interpreted code. While it executes fairly efficiently, it is nevertheless interpreted and will never execute as efficiently as compiled code. This should not present any concern for HMI/SCADA applications since IWS is performing the real-time management of the tag database and key functions such as alarming, logging, etc. The interpreted nature of VBScript allows changes to be made quickly to an application. IWS supports dynamic, on-line configuration and this capability is maintained with the addition of VBScript support

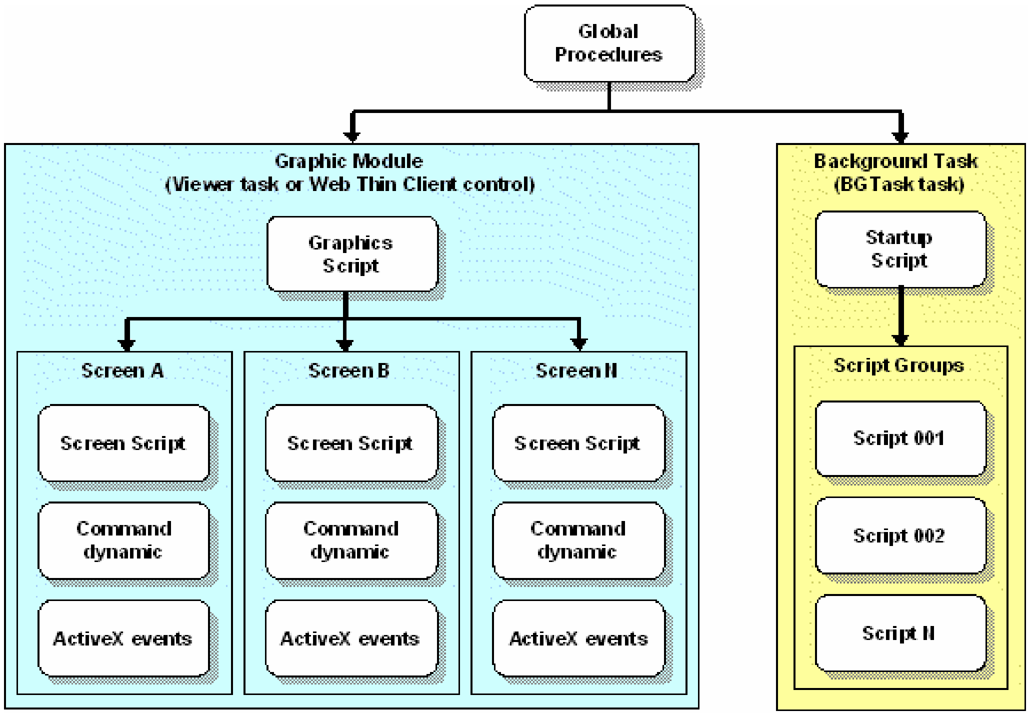
Developers familiar with IWS know that in the bottom left corner of the development window are tabs that provide access to the Database, Graphics, Tasks, and Communications Workspace folders containing the different application components. The developer will need to navigate among these different folders and application components when using VBScript.

VBScript interfaces can be found in 6 different areas:

- Database Workspace folder – Global Procedures
- Graphics Workspace folder - Graphics Script
- Graphic Screens – Screen Scripts
- IWS Objects on a Screen – Command Dynamic
- ActiveX Objects on a screen – ActiveX Events
- Tasks Workspace folder – Background Startup Script and Background Script Groups

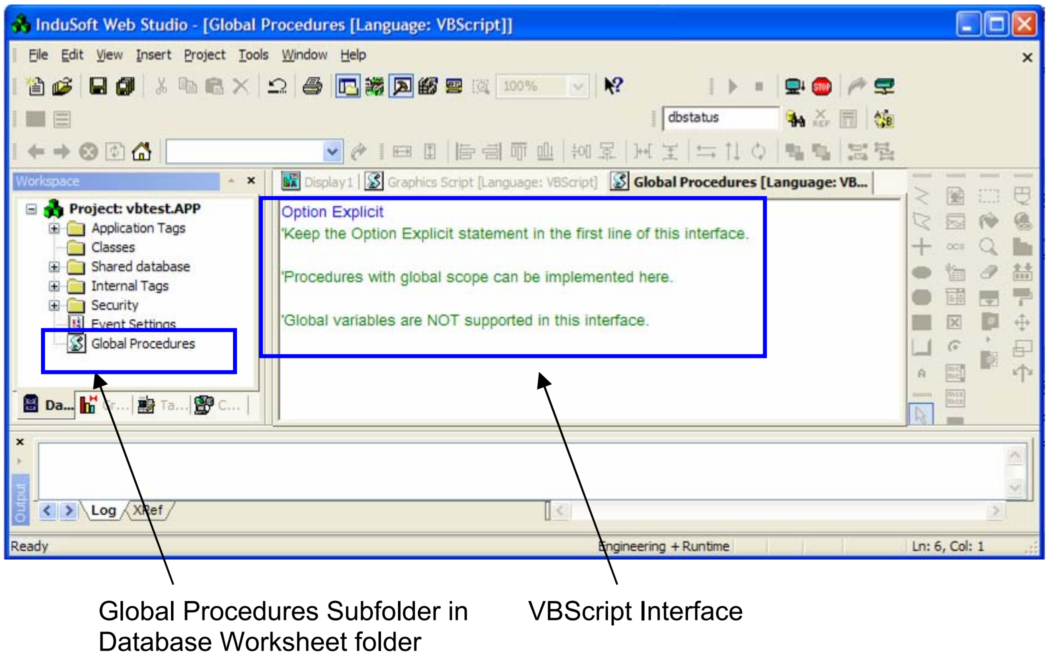

Subfolders and Icons within a Workspace folders
Tabs to access various Workspace folders
InduSoft Web Studio - [Display1]
File Edit View Insert Project Tools Window Help
100%
dbstatus
Workspace
Project: vbtest.APP
Screens
Group Screen
Web Pages
Symbols
Graphics Script
Library
Layout
Display1
Log
XRef
Engineering + Runtime
X: 176, Y: 443

The figure below shows the structure of the VBScript interfaces within a typical IWS project (application). Note that there are certain types of VBScript interfaces that have one instance (e.g. Global Procedures, Background Startup Tasks and Graphic Script) while others can have multiple instances (e.g. Background Script Groups, Screen Scripts, Command Dynamic and ActiveX Events).



Global Procedures

Global Procedures are located in the Database Worksheet folder. Global Procedures are shared by both the Graphics Module Scripts (Graphics Script and Screen Scripts) and the Background Task Scripts (Background Startup Script and Background Script Groups). Note that it this is only the Procedures that are shared, not the Variables. Other VBScript interfaces within the Graphic Module or Background Task do not share variables or procedures between them; they are independent of each other.


InduSoft Web Studio - [Global Procedures [Language: VBScript]]
File Edit View Insert Project Tools Window Help
Workspace
Project: vbtest.APP
Application Tags
Classes
Shared database
Internal Tags
Security
Event Settings
Global Procedures
Display1
Graphics Script [Language: VBScript]
Global Procedures [Language: VB...
Log
XRef
Ready
Engineering + Runtime
Ln: 6, Col: 1

```
Option Explicit
'Keep the Option Explicit statement in the first line of this interface.

'Procedures with global scope can be implemented here.

'Global variables are NOT supported in this interface.
```

Global Procedures Subfolder in Database Worksheet folder

VBScript Interface

Key Notes:

- **Before executing the application, be sure to save (or close) the Screen after any VBScript is entered. Otherwise the changes might not be updated. This is true for all VBScript interfaces.**

Graphics Script

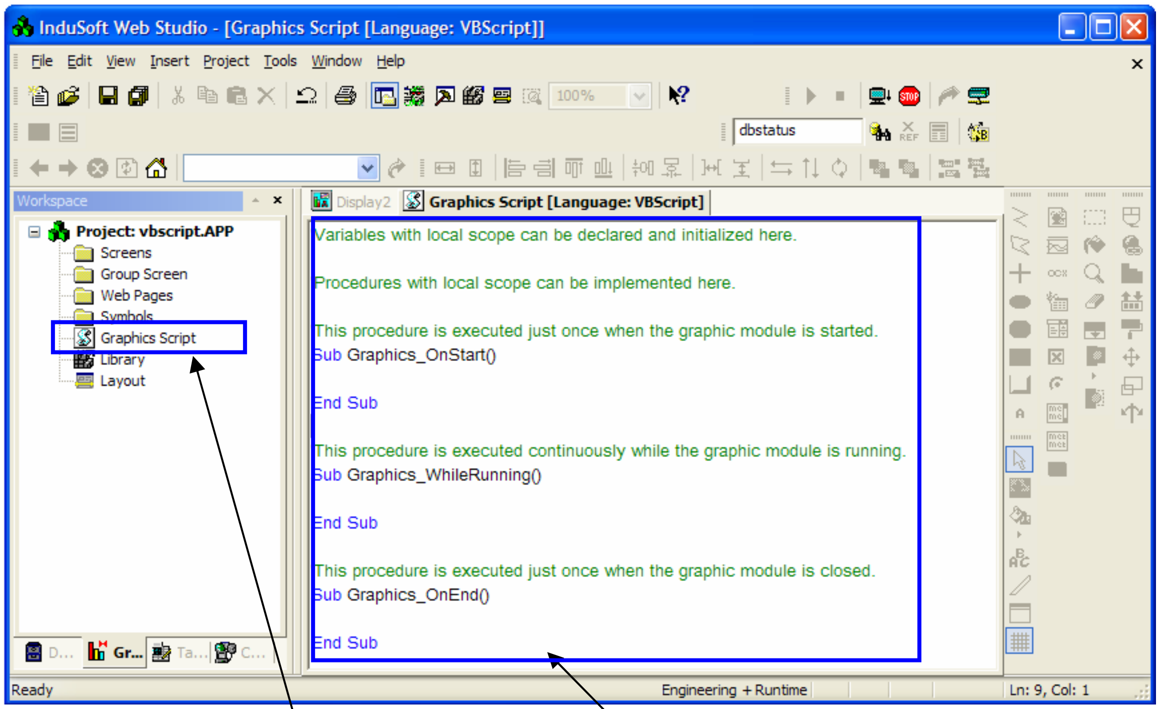
The Graphics Script is located in the Graphics Worksheet folder. Procedures and Variables declared in the Graphics Script interface are available locally but are not accessible by any Screen Script interface, or from any other VBScript interface within IWS. Procedures and Variables declared in a Screen Script interface are not accessible by the Graphics Script. If common Procedure(s) are required, they should be put into the Global Procedures interface. Note that the Graphics Script is scanned (processed) by IWS before the Screen Scripts.

The Graphics Script has three different pre-configured subroutines to execute VBScript code. These subroutines execute the VBScript contained in them based on the event state of the Graphics Module. These are:

Graphics_OnStart
Code contained within this subroutine is automatically executed just once when the Graphics Module is started. This is a good area to initialize variables or execute start-up code.

Graphics_WhileRunning
Code contained within this subroutine is automatically executed continuously while the Graphics Module is running. The rate at which this subroutine is called depends on the performance of the hardware platform and other tasks running at the time.

Graphics_OnEnd
Code contained within this subroutine is automatically executed just once when the Graphics Module is closed.



```
Variables with local scope can be declared and initialized here.

Procedures with local scope can be implemented here.

This procedure is executed just once when the graphic module is started.
Sub Graphics_OnStart()

End Sub

This procedure is executed continuously while the graphic module is running.
Sub Graphics_WhileRunning()

End Sub

This procedure is executed just once when the graphic module is closed.
Sub Graphics_OnEnd()

End Sub
```

Graphic Script Icon in
Graphics Worksheet folder

VBScript Interface

Key Notes:

- **Do not change the name of the pre-configured subroutines in the VBScript interface. Otherwise they many not properly execute.**

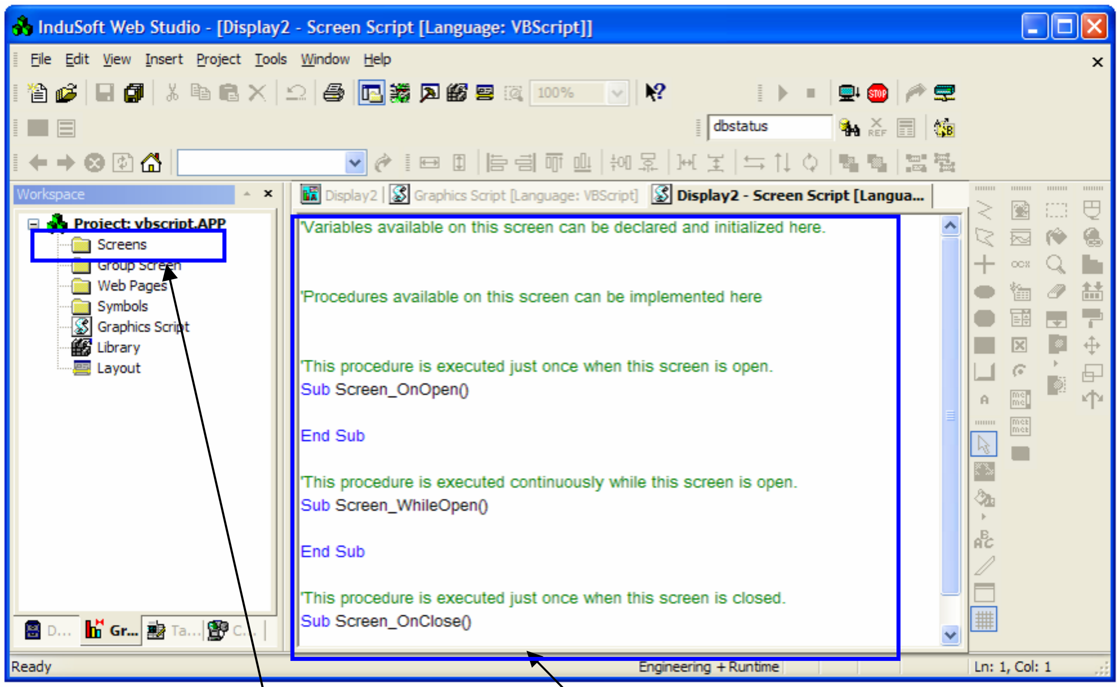
The Graphics Script operates for both the Server (the host processor where the IWS application is running) and Web Thin Clients (web browser interface using Microsoft Internet Explorer). For the Server, the Graphics module is the Viewer task (the display on the host processor), while the ISSymbol control is the Graphics module for Web Thin Clients.

The operation of the Graphics Script on the Server is described above, and starts when the application is started on the Server, assuming there are one or more screens. But since Web Thin Clients can log on at any time after the Server is started, the functioning of the Graphics Script is different for Web Thin Clients and is independent of the operation of the Graphics Script on the host Server. Web Thin Client operation is as follows:

- When a Web Thin Client logs on to the Server, following completion of the log on process, the Graphics_OnStart subroutine will be executed for the Web Thin Client. This will occur each time any new Web Thin Client logs on to the Server.
- Following completion of the execution of the Graphics_OnStart subroutine, the Graphics_WhileRunning subroutine will be executed for as long as the Web Thin Client (browser) hosts the ISSymbol control (i.e. while an active network link exists and the ISSymbol is active in the browser).
- When the Web Thin Client is shut down or when the ISSymbol control is no longer hosted by the browser, the Graphics_OnEnd subroutine is executed

Screen Scripts

Screen Scripts are associated with individual graphical screens. These screens can be for display on the host Server (where the IWS application is running), for a Web Thin Client, or both. Procedures and Variables declared in a Screen Script VBScript interface are not accessible by any other VBScript interface within IWS. However, the Screen Script interface can access procedures declared in the Global Procedures script interface.

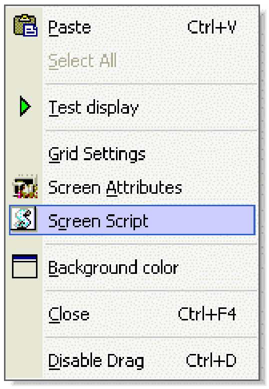

InduSoft Web Studio - [Display2 - Screen Script [Language: VBScript]]
File Edit View Insert Project Tools Window Help
dbstatus
Workspace
Project: vbscript.APP
Screens
Group Screen
Web Pages
Symbols
Graphics Script
Library
Layout
Display2
Graphics Script [Language: VBScript]
Display2 - Screen Script [Langua...
Ready
Engineering + Runtime
Ln: 1, Col: 1

```
'Variables available on this screen can be declared and initialized here.


'Procedures available on this screen can be implemented here


'This procedure is executed just once when this screen is open.
Sub Screen_OnOpen()

End Sub

'This procedure is executed continuously while this screen is open.
Sub Screen_WhileOpen()

End Sub

'This procedure is executed just once when this screen is closed.
Sub Screen_OnClose()
```

Screen subfolder in Graphics Worksheet folder

VBScript Interface

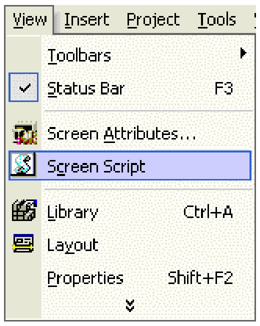
There are two methods to access a Screen Script. The first is to select the desired Screen and have it displayed on the active IWS workspace. Then, perform a right mouse click while the cursor is located on the display screen. A pop-up menu will let you select the **Screen Script** (as shown at the right). When the **Screen Script** option is selected, the IWS workspace will display the Screen Script VBScript interface.



Pop-Up Menu

Notice the Screen Script VBScript interface looks very similar to the Graphics Script interface. The differences between the Screen Script and the Graphics Script are:

- There is only one Graphics Script. The Graphics Script is activated when the Graphics Module starts
- You can have multiple Screen Scripts. There is one Screen Script available per Screen, but you can have multiple screens.



Pull-down Menu

The second method to access a Screen Script is to select the desired Screen and have it displayed on the active IWS workspace. Then from the top toolbar, select **View**. A pull-down menu (as shown at the right) will have the **Screen Script** option available. By selecting this option, you will activate the Screen Script VBScripting interface.

The Screen Script interface has three predefined subroutines. These are:

Screen_OnOpen
Code contained within this subroutine is automatically executed just once when the Screen is opened.

Screen_WhileOpen
Code contained within this subroutine is automatically executed continuously while the Screen is open. The rate at which this subroutine is called depends on the performance of the hardware platform and other tasks running at the time.

Screen_OnClose
Code contained within this subroutine is automatically executed just once when the Screen is closed.

The execution of the Screen Script subroutines on the Server executes independently from the execution on Web Thin Clients.

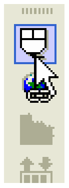
Key Notes:

- **Do not change the name of the pre-configured subroutines in the VBScript interface. Otherwise they many not properly execute.**
- **Before executing the application, be sure to save (or close) the Screen after any VBScript is entered. Otherwise it might not be updated. This is true for all VBScript interfaces.**
- **The Graphic Script is scanned (processed) by IWS before the Screen Scripts are processed.**

Command Dynamic

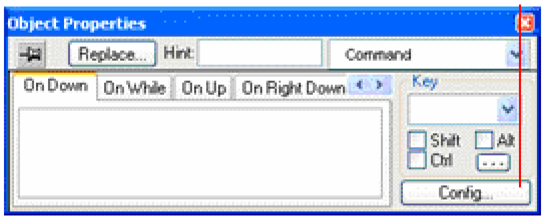
A Command Dynamic is associated with a specific object on a Screen, and allows one or more actions to take place when an event occurs with the specific object. A typical use is a button (perhaps a rectangle) that is placed on the screen. When an operator selects on the button (via mouse click or pressing a touchscreen over the object), this action is expected to initiate some action. That action may be to set/reset a PLC bit, jump to a different screen, whatever. The Command Dynamic allows the developer to chose what action to take.

With Version 6.1, IWS adds new capability to the Command Dynamic interface. In addition to the IWS built-in language command, the Command Dynamic can now execute VBScript code. The steps to access the VBScript interface within a Command Dynamic are:



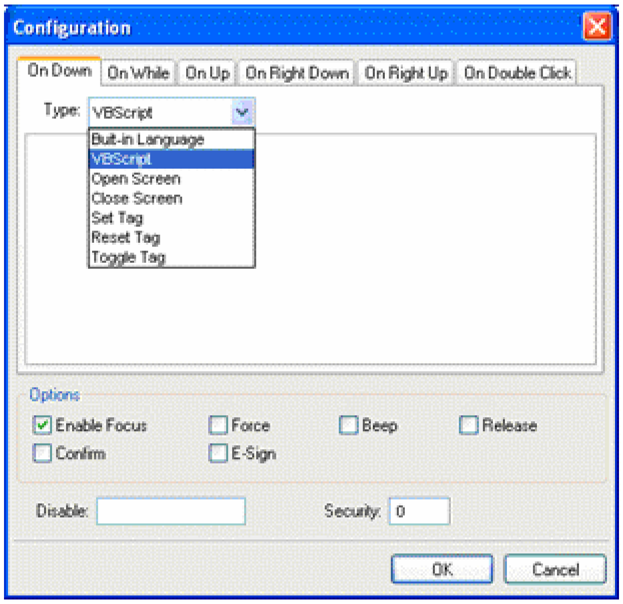
1. Select the object on the Screen currently opened in the IWS workspace. If the object has a Command Dynamic associated with it, then right click on the object. Otherwise, click on the Command Dynamic icon (right) from the Mode toolbar and then right click on the object.

2. Now, the Object Properties dialog box for the Command Dynamic will open. Click on the Config… button in the lower right corner of the dialog box.



3. Select the event condition (e.g. On Down) where your want code to be execute and then select VBScript as the Type.

4. Enter your VBScript code (variable declarations and executable statements).



Within the Command Dynamic, you enter VBScript variables and executable statements subject to the following conditions:

- Any variable declared in this interface will only have a local scope.
- You cannot implement procedures (i.e. Subroutines or Functions) within this interface.

Notwithstanding these restrictions, VBScript code within a Command Dynamic still has access to all Global Procedures.

VBScript code within the Command Dynamic interface is executed whenever one or more of the selected event conditions (listed in the Command Dynamic configuration screen) occur for the selected object. The execution of the Command Object script on the Server executes independently from the execution on Web Thin Clients.

Key Notes:

- **Before executing the application, be sure to save (or close) the Screen after any VBScript is entered. Otherwise it might not be updated. This is true for all VBScript interfaces.**

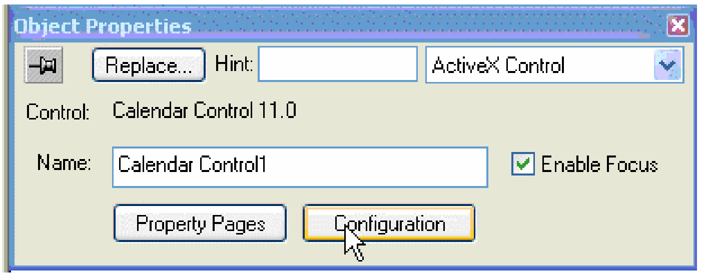
ActiveX Events

IWS is an ActiveX container, supporting ActiveX controls, generally inserted on a given graphical screen. With IWS Version 6.1, there is a VBScript interface to ActiveX Events so that an ActiveX object event can trigger a VBScript code segment.

The steps to accessing the VBScript ActiveX Event interface are as follows:

1. Select the ActiveX object on the Screen currently opened in the IWS workspace. Right click on the object to open its Object Properties dialog box. If you need to insert an ActiveX object, select the ActiveX Control icon from the Mode toolbar and then right click on the object

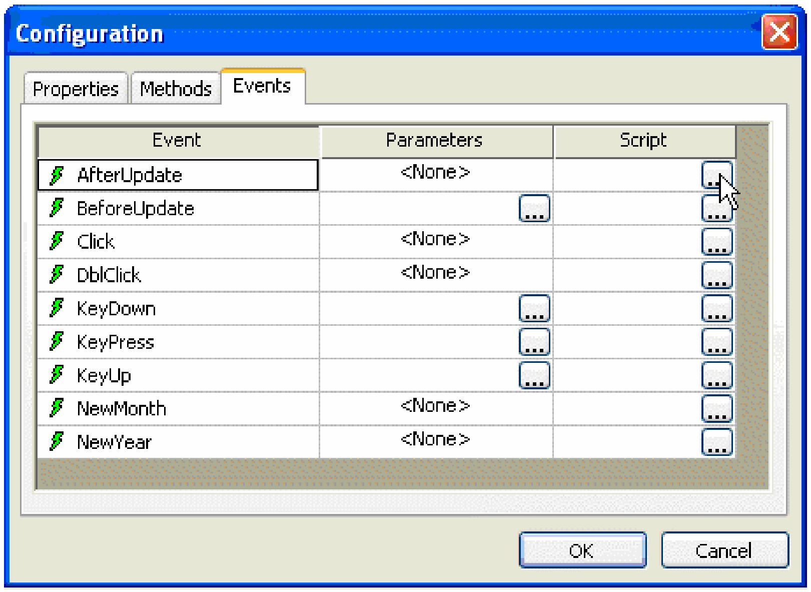
   In the lower right corner of the ActiveX Object Properties dialog box will be a Configuration button. Click this to open up the Configuration options dialog box.



Hint:
ActiveX Control
Control: Calendar Control 11.0
Name: Calendar Control1
☑ Enable Focus

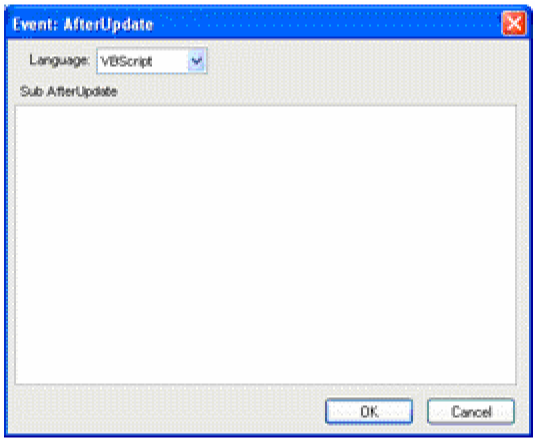
2. Click on the Events tab (as shown at the right).

3. Click on the … button in the Script Column for the event you want to write VBScript for.



Event	Parameters	Script
AfterUpdate	<None>	...
BeforeUpdate	...	...
Click	<None>	...
DblClick	<None>	...
KeyDown	...	...
KeyPress	...	...
KeyUp	...	...
NewMonth	<None>	...
NewYear	<None>	...

This is the scripting interface for ActiveX Events. Be sure VBScript language is selected. You can now insert code that will execute when the selected ActiveX Event is triggered.



Language: VBScript
Sub AfterUpdate

Within the ActiveX Event interface, you enter VBScript variables and executable statements subject to the following conditions:

- Any variable declared in this interface will only have a local scope.
- You cannot implement procedures (i.e. Subroutines or Functions) within this interface.

Notwithstanding these restrictions, VBScript code within the ActiveX Event interface still has access to all Global Procedures, <u>as well as any procedures in the Screen Script</u> for the same Screen where the ActiveX object is configured.

VBScript code within the ActiveX Event interface is executed whenever one or more of the selected Event conditions (listed in the Configuration dialog box) occur for the selected ActiveX object. The execution of the script on the Server executes independently from the execution on Web Thin Clients.

Key Notes:

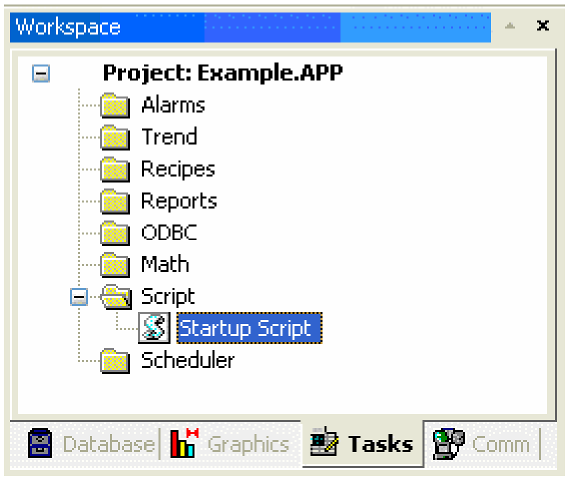
- **Before executing the application, be sure to save (or close) the Screen after any VBScript is entered. Otherwise it might not be updated. This is true for all VBScript interfaces.**

Background Task Startup Script

In the Tasks Worksheet folder is the Script subfolder which will contain a default Startup Script icon and any Background Task Script Groups declared. To edit the Background Task Startup Script:

1. Click on the Tasks Worksheet folder

2. Click on the Script subfolder.



Any VBScript code placed in this interface will execute when the Background Task module is started, which occurs when the IWS application is started. This code will only execute once, and is meant for initialization purposes.

Variables and Procedures declared in the Background Task Startup Script are available to the Background Task Script Group, but are not available to any VBScript interfaces in the Graphic Module. Remember that the Background Task Group Startup Script can access the procedures declared in Global Procedures.

Since the Background Task Startup Script has no interaction with a Graphics script, the only Server display I/O functions that can be implemented are **MsgBox** and **InputBox** functions.

Since the Background Task Startup Script runs on the IWS Server, there is no effect with Web Thin Clients.

<u>Background Task Script Groups</u>

The Background Task Script Groups consist of one or more VBScript interface groups that run in the Background Task. By default, there are no Background Task Script Groups unless added by the developer. These Script Groups will execute in a background as long as their Execution Field is in a **TRUE** state.

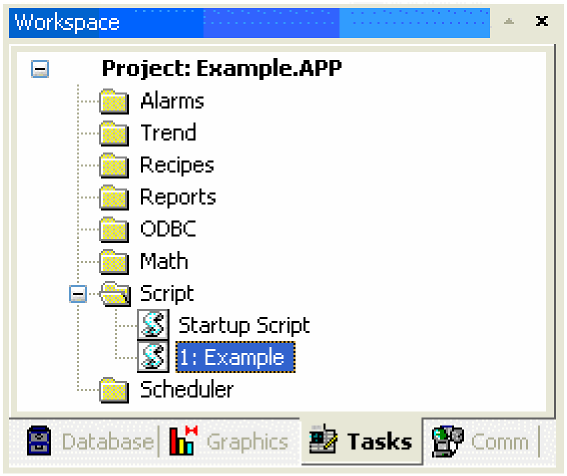
Background Task Script Groups have the following limitations:

- Variables declared in a Background Task Script Group have a local scope for it's specific Script Group only. Variables cannot be shared with other Script Groups, nor any other VBScript Interface.
- Background Task Script Groups cannot declare their own Procedures (Subroutines and Functions).
- The Execution Field of the Script Group <u>will only support IWS tags or built-in functions</u>. No support for VBScript variables or Procedures is provided in the Execution Field.

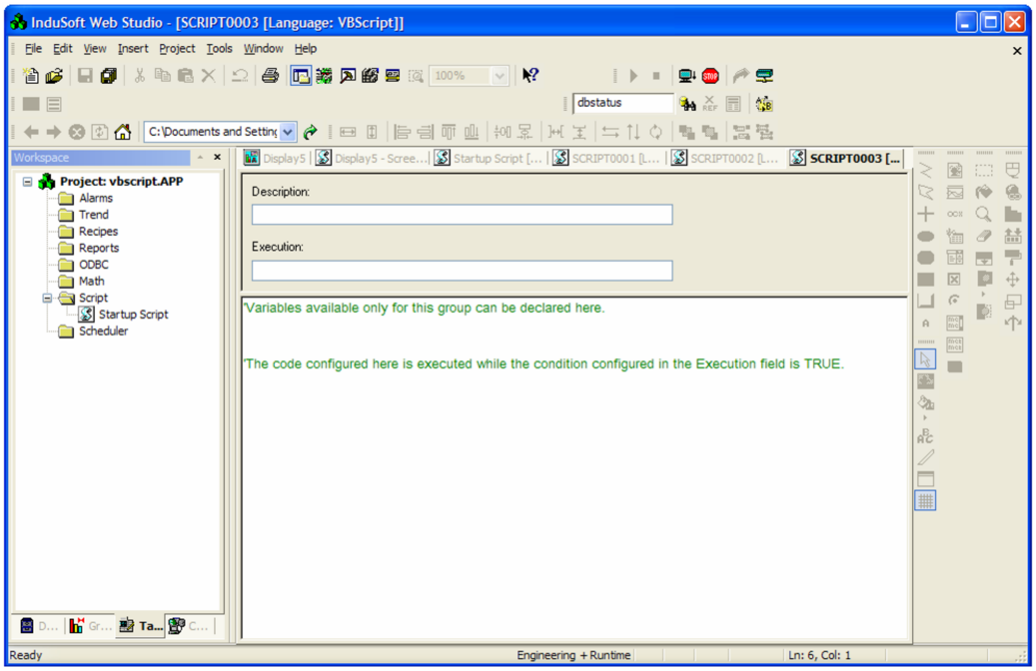
However, the Background Task Script Groups can do the following:

- Access Procedures and Variables within the Background Task Startup Script.
- Access Procedures declared in Global Procedures.

To create a new Script Group, right-click on the Script subfolder in the Tasks tab of the Workspace. Select the Insert option from the pop-up menu. Note that the Startup Script is already defined. To open (edit) an existing Script Group, simply click its icon in the Script subfolder of the Tasks workspace tab.



The code configured in each Script Group is executed by the Background Task. IWS scans the Script Groups sequentially (based on the number of the group) and executes only the Groups in which the condition configured in the Execution Field of the Script Group is set to or is evaluated to be **TRUE** (a value different from 0).



'Variables available only for this group can be declared here.

'The code configured here is executed while the condition configured in the Execution field is TRUE.

When any Script Group is saved during runtime (e.g. from an on-line configuration download), the Startup Script interface will be executed again, and the current value of the local variables contained in any Script Group will be reset, if any exist.

Since the Background Task Script Groups run on the IWS Server, there is no effect with Web Thin Clients.

Key Notes:

- **The Execution Field of the Script Group only supports syntax as specified by the IWS built-in language.**
- **Before executing the application, be sure to save (or close) the Screen after any VBScript is entered. Otherwise it might not be updated. This is true for all VBScript interfaces.**
- **If any Script Group is saved during runtime (i.e. on-line configuration), the Startup Script Group will be executed again and the current value of local variables will be reset**

Scope of VBScript Procedures and Variables

The following table summarizes the relationship between the IWS VBScript interface location and its Scope of Variables and Procedures. The table also defines where the Scripts are located

Item	Scope of Procedures and Variables	Execution	Functionality	Location/Access
Global Procedures	All Procedures are global, Variables are accessible only within Global Procedures (local)	Procedures are accessible to any Script on the host Server	Declaration of Procedures (Functions and Subroutines) that are available globally	Database Workspace Folder
Graphics Scripts	Procedures and Variables accessible within Graphics Script interface only. Can Call Global Procedures.	Executes on host Server and/or Web Thin Client where any screen is displayed.	Condition-based execution - Graphics Start - Graphics Open - Graphics Close Graphics Scripts execute before Screen Scripts	Graphics Workspace Folder
Screen Scripts	Procedures and Variables accessible within Screen where the Script is written. Screen Script procedures accessible to ActiveX Events for ActiveX objects contained in the Screen. Can Call Global Procedures.	Executes on host Server and/or Web Thin Client where the specific screen is displayed	Condition-based execution - Screen Start - Screen Open - Screen Close	Within the Screen.
Command Dynamic	Variables and Script accessible only in Object where the Script is configured. Can Call Global Procedures.	Executes on host Server and/or Web Thin Client where the screen with the specific Object is displayed	Execution of Script when Object condition is met	Within Object (Command) Properties. The Screen that uses the Object must be open.
ActiveX Events	Variables accessible only in Object where the Script is configured. Screen Script Procedures are accessible. Can Call Global Procedures.	Executes on host Server and/or Web Thin Client where the screen with the specific Object is displayed	Execution of Script when selected ActiveX Event occurs	Within the ActiveX object. The Screen that uses the Object must be open.
Background Startup Script	Procedures and Variables accessible within the Script Group. Can Call Global Procedures.	Executes on Server as a background task	Declaration of Procedures and Variables that are available for Background Scripts	Tasks Workspace Folder
Background Script Groups	Accessible within Script Group only. Can Call Global Procedures.	Executes on Server as a background task	Condition-based execution in background mode. Can have multiple Script pages.	Tasks Workspace Folder

Accessing IWS Tags and IWS Built-in functions

When writing your code in a VBScript interface, you can access any tag from the IWS tags database or any function from the IWS built-in language by applying the "$" prefix to the tag/function name, as in the examples below:

```
CurTime = $Time                ‘ Returns the value of the tag Time from the tags database
a = $MyTag                     ‘ Sets a to the value of the IWS tag MyTag
$Open("main.scr")              ‘ Executes the Open() function to open the "main" screen
```

IWS tags and built-in functions are accessible from any VBScript code segment, regardless where located. If the IWS function returns a value (e.g. error or status information), this can be assigned to a VBScript variable. IWS tags can be used as arguments in VBScript statements and functions.

If an undefined name follows the “$”, when the programmer does a **Check Script** function or attempts to **Save** the script, IWS will ask the programmer if they want to define the IWS tag, and if so, prompt for the tag type.

IWS supports the following application tag types:

- **Boolean** (a Boolean (True/False) or digital value (0 or 1))
- **Integer** (a 32-bit long-word signed integer type)
- **Real** (a real number stored as a double precision word)
- **String** (a string of characters of up to 255 characters that holds letters, numbers, or special characters)
- **Class** (a user-defined, compound tag)
- **Array** (an array of values from 0 to 16,384)

Passing variables between VBScript and IWS is straightforward but there are some conversion considerations that should be noted:

IWS Boolean

With VBScript, variable can be of the data subtype Boolean. VBScript defines keywords **True** and **False** for logical states True and False, respectively. In VBScript, False has a numeric value of 0, while True has a numeric value of -1. This is because Booleans are not actually stored as bits, but as 32-bit signed integers. If all bits are zero, then it is a 0 or logical False. If all bits are set to 1, then it is a signed value of -1 or a logical True.

IWS objects that display IWS-defined boolean tags (e.g. Text I/O) will have the boolean values displayed as 0 or 1 (0=False, 1=True), not as False or True. Consider the following VBScript code segment:

```
$MyBool = True           ‘ Will be displayed as a “1” in an IWS object (*see below)
$MyBool = False          ‘ Will be displayed as a “0” in an IWS object
```

The value for True assumed by Boolean IWS tags depends on the value of the parameter **BooleanTrueAboveZero** that is located in the **[Options]** section of the <Application>.APP file. To access this parameter, you need to open the <Application>.APP file with a simple text editor such as Microsoft Notepad. For example:

```
[Options]
BooleanTrueAboveZero = 0        IWS Boolean tag set to value 1 (True) when value <> 0

[Options]
BooleanTrueAboveZero = 1        IWS Boolean tag set to value 1 (True) only when value > 0
```

One item to watch for is the boolean NOT operator. With an IWS tag, even though the tag is of type Boolean, it is really stored internally as a 32-bit signed variable. If you NOT a 0, the lower bit is set to one but in reality all the bits are set to 1's, meaning that with a variable that is a signed integer, the NOT of 0 is really -1. For example,

```
   a = CBool(Not(0))
   $c = a                          ' $c (IWS tag c) will display as  -1
```

One programming trick that can be used when attempting to toggle IWS Boolean Tags between 0 and 1 is either:

```
    $tag = Abs ($tag=0)            ' Either one of these statements will toggle the tag
    $tag = $If ($tag=0,1,0)        ' between 0 and 1
```

IWS Integer

All IWS integer tags are stored as 32-bit values. VBScript has 3 different variant subtypes that are of interest. Bytes are 8-bit values that are positive whole numbers ranging from 0 to 255. Integers are 16-bit signed values that range from -32,768 to 32,767. Long Integers are 32-bit values that range from -2,147,483,648 to 2,147,483,647.

When storing to an IWS integer tag, the conversion to a 32-bit signed integer type will be automatically made. For example:

```
    a = CInt (-30)                 ' a is a 16-bit signed integer with a value of -30
    $MyInt = a                     ' MyInt is a 32-bit signed integer with a value of -30
    b = CByte (-30)                ' Generates an error since Bytes are 0 to 255, not negative
    b = CByte (30)                 ' b is a 8-bit unsigned integer with a value of 30
    $MyInt = b                     ' MyInt is a 32-bt signed integer with a value of 30
```

When converting from an IWS integer tag to an IWS tag, this is really not a problem since VBScript variables are type variant. For example:

```
    $MyInt = 400                   ' Store a vale larger than 255 (the Byte limit)
    a = CByte (10)                 ' store as a byte subdata type
    a = $MyInt                     ' a will equal 400.
```

IWS Strings

In IWS, strings are up to 255 in length, while VBScript strings can be virtually unlimited in length (limited by available memory only). During the conversion from a VBScript string variable to an IWS string, any characters beyond the first 255 will be truncated. For example:

```
    a = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
    a = a & a & a & a & a & a & a & a & a & a   ' String is 360 characters long
    $MyStr = a                     ' Store string in IWS string
                                   ' Result is 7 strings of a + "ABC" for total of 255 characters
```

In most cases, this string length difference is not of material significance. However, certain ActiveX Controls can be used for block transfer of data to real-world devices and strings are ideal for forming variable length data blocks. The string can then be parsed to extract the data of interest.

IWS Classes

IWS Classes are simply user-defined compound tags that can be made up of one or more IWS tag type. The IWS Classes and Tags are defined in the Database Worksheet. For example, if we define a IWS Class (under the Classes Folder in the Database worksheet) called MyClass with the following elements

```
MyClass
    Item1      Integer
    Item2      Integer
    Message    String
```

Next, a Class tag is created (in the Application Tags Folder)

```
Cls1    MyClass
```

Finally, in VBScript, we can refer to the elements in the Class tag as follows:

```
$Cls1.Item1 = 10
$Cls1.Item2 = 20
$Cls1.Message = "Hello World"
```

IWS Arrays

Using the Class example from above, if (in the Application Tags folder) we had declared the variable Cls1 to have a size of 10, this would be an array with 11 elements. [Remember that the count starts at 0, not 1].

In VBScript, we would refer to the elements in the Class array tag as follows:

```
$Cls1[1].Item1 = 10
$Cls1[1].Item2 = 20
$Cls1[1].Message = "Hello World"
```

We can also use a VBScript variable for the index of the Class array tag. For example:

```
Dim i
i = 1
$Cls1[i].Item1 = 10
$Cls1[i].Item2 = 20
$Cls1[i].Message = "Hello World"
```

Key Notes:

- **IWS tags can be added through the VBScript interface. Simply type a "$" followed by a valid IWS name, and when the Script is Saved, Closed or Check Script function invoked, the programmer will be prompted to create new IWS tag(s).**
- **VBScript variables and IWS variables can be passed to each other.**
- **Watch for string length differences with IWS (max. 255 characters) versus VBScript (no limit).**

Accessing ActiveX Objects from VBScript

Any of the VBScript interfaces relating to a Screen (i.e. Screen Script, Command Dynamic, and/or ActiveX Events) can directly access the Properties and Methods of an ActiveX control (OCX) that is inserted on a screen.

Using ActiveX Controls is fairly straight forward. First, the ActiveX controls must be registered (i.e. the Operating System Windows Registry must have an entry and Class ID (CLSID) established for the ActiveX Control). Usually when an ActiveX Control is installed in the PC, the installation program will register the ActiveX Control in the final stage of the installation process. If not, registration can be done with one of two methods:

1. Use the Microsoft **RegSvr32** command
   - Invoke the Microsoft Windows **Run** command
   - In the dialog box, type CMD, then OK
   - Type REGSVR32 "C:\<path to OCX control>\<ActiveX Control Name>.OCX", then Enter (Be sure path name is in quotes)
   - If the ActiveX Control registers properly, you will get a message indicated this
   - Close the dialog box
2. Use the **Register Controls** utility provided by IWS (under **Tools** on the main toolbar)
   - Click on **Tools,** then **Register Controls**
   - On the dialog box that pops up, click on **Register**
   - Use the file navigator to locate the ActiveX Control that you want to register
   - Click on Open.
   - Click on Close in the Register Controls dialog box.

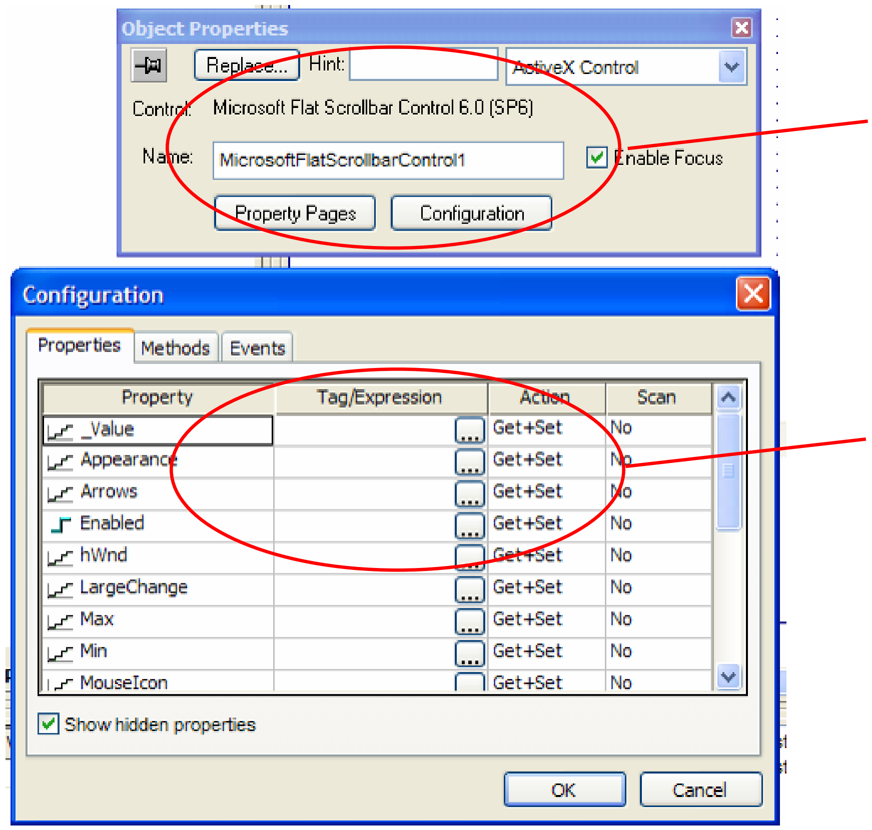
You can also use the IWS **Register Controls** utility to verify that the ActiveX Control has been registered. Beware that the registered name and the file name may not be the same, and in many cases they are not. The best way to verify the control is properly registered is to examine the path of the registered Control.

When the ActiveX Control has been registered, it can be inserted onto a display screen using either the OCX tool in the IWS toolbar or by using the Insert -> ActiveX Object from the top toolbar. A dialog box will appear with a scrolled list of ActiveX objects that are available. Insert the ActiveX object(s) that are appropriate for the application by clicking on OK. IWS will automatically assign a unique name to the ActiveX control. You can use this name or change it, the only requirement being that it must be unique from other ActiveX controls.

Now that an ActiveX Control has been placed on the Screen, any VBScript interface associated with that screen can access the ActiveX Control. These VBScript interfaces are limited to the Screen Script, Command Dynamic for objects located on the same Screen, and ActiveX Event Handler for other ActiveX objects located on the same Screen.

For example, Microsoft has an ActiveX scrollbar control called "MicrosoftFlatScrollBarControl 6.0 (SP6)". Assuming this was inserted for the first time onto a Screen in an IWS application, IWS would likely name this Control "MicrosoftFlatScrollbarControl1". For brevity, let us rename this to "MFSC1". I could easily click on the ActiveX Control on the screen to access its Property Pages, Properties, Methods and Events.

Note that Property Pages and Events are not accessible through the VBScript Interface, although a VBScript Interface is available with the ActiveX's Event Handler. Only an ActiveX Control's Properties and Methods are available from VBScript as implemented in IWS. By clicking on the object to get the Object Properties dialog box



Hint:

ActiveX Control

Control: Microsoft Flat Scrollbar Control 6.0 (SP6)

Name: MicrosoftFlatScrollbarControl1

[✓] Enable Focus

Property	Tag/Expression	Action	Scan
_Value		Get+Set	No
Appearance		Get+Set	No
Arrows		Get+Set	No
Enabled		Get+Set	No
hWnd		Get+Set	No
LargeChange		Get+Set	No
Max		Get+Set	No
Min		Get+Set	No
MouseIcon		Get+Set	No

[✓] Show hidden properties

ActiveX Control Name established by IWS. You can rename this Control.

Use this interface to tie Properties and Methods to IWS tags.

To access the ActiveX Control's Properties and Methods from VBScript, you simply type the name of the Control, followed by a Period "**.**" and then the Property or Method. You will need to reference documentation from the developer of the ActiveX Control to determine which properties are setting (Set) or retrieving (Get), and the functioning of the Methods available. For example, with the Microsoft scroll bar control, we access Properties using the following code:

```
    MFSC1.Min = 0                    ' Set the min value of the scroll bar to 0
    MFSC1.Max = 100                  ' Set the max value of the scroll bar to 100
    $LocTag = MFSC1.Value            ' Get the current location of the scroll bar, pass to IWS tag
```

IWS tools such as **Position** and **Command** can be used with ActiveX controls. To enable these tools, insert the ActiveX control on the Screen and then make sure the ActiveX control is selected (highlighted). Then, select the **Position** or **Command** tool. For example, with the **Position** tool, you can control the visibility of the ActiveX Control, or change its location on the screen.

Key Notes:

- **All ActiveX Controls must have a unique name**
- **When referencing an ActiveX object name that has been inserted on a screen, note that the reference is case-sensitive from VBScript.**
- **Only ActiveX Properties and Methods can be accessed via VBScript. Event handling must be set-up by configuring the object (i.e. right click on the object)**
- **ActiveX Controls can only be accessed by VBScript interfaces associated with the Screen which contains the ActiveX Control (i.e. Screen Script, Command Dynamic, ActiveX Event Handler)**

IntelliSense

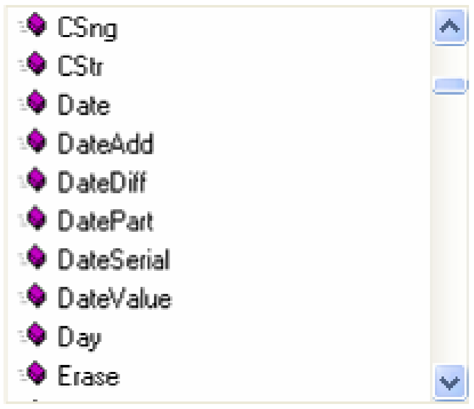
The VBScript Editor provides a useful tool called IntelliSense, a feature first popularized in Microsoft Visual Studio. Intellisense can be thought of providing "auto-completion" based on the language elements, variables and class members, as well as a convenient listing of available functions. As the developer

IntelliSense the dialog box can display the following:

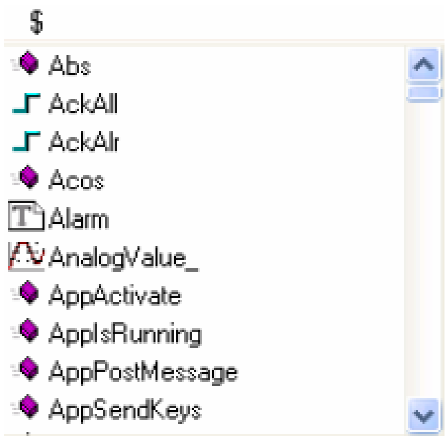
- VBScript Functions
- ActiveX Controls, Properties and Methods (the ActiveX Control must be inserted on the Screen where the Screen Script, Command Dynamic or ActiveX Event is used)
- IWS tags and tag fields.
- IWS built-in functions

As the programmer begins to type and characters are recognized, IntelliSense may turn on. If not, the programmer can activate IntelliSense by pressing the Ctrl key plus the Spacebar ("Ctrl" + " "). By typing a "$" at the beginning of a line, this allows access to IWS tags and built-in functions to be referenced.

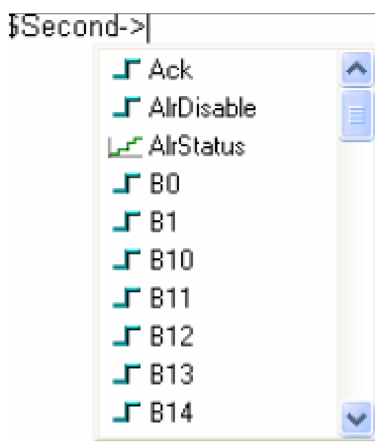
When IntelliSense is activate, a pop-up box will appear. The contents of the pop-up box depend on what the programmer has already typed. Sample IntelliSense pop-up dialogs are shown below:



IntelliSense Dialog for VBScript Functions



IntelliSense Dialog for IWS Functions



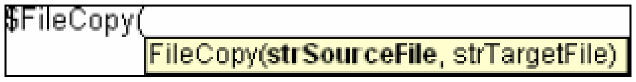
IntelliSense Dialog for IWS Tag Fields

Note that VBScript variables are not accessible through the IntelliSense dialog box.

IntelliSense uses different Icons to indicate the type of item that is being referenced. Some Icons are used to indicate different items, so it is important to notice what object is being referenced (i.e. is it an IWS tag, ActiveX Control, VBScript function, etc.)

IntelliSense Icon	Use
⌐	IWS Boolean Tag
📈	IWS Integer Tag
〰	IWS Real Tag
T	IWS String Tag
📁	IWS Class Tag
◆	VBScript Function, built-in IWS function, or ActiveX Control Method
◈	ActiveX Control Property, VBScript Constants

For many of the functions (both VBScript functions and IWS built-in functions), IntelliSense provide a Parameter Quick Info pop-up dialog. This pop-up dialog may appear once the VBScript or IWS function is entered. An example is:


```
$FileCopy(
FileCopy(strSourceFile, strTargetFile)
```

Key Notes:

- **Use the Ctrl key plus Spacebar key (“Ctrl” + “ ”) to activate IntelliSense. Doing this on a blank line will show all available VBScript functions and any ActiveX controls available.**
- **Use the Ctrl key plus Spacebar key (“Ctl” + “ ”) to auto-complete any VBScript function, IWS tag, IWS tag field, IWS Class or Class Member, IWS built-in function, or ActiveX Control name, Property or Method once enough of the characters have been entered so that the reference is no longer ambiguous.**
- **Typing a “$” at the beginning of a line will invoke IntelliSense, referencing existing IWS tags and built-in functions**
- **Typing the name of an IWS tag, followed by the minus key “-“ plus a greater than arrow key “>” will open the list of available fields for the IWS tag**

VBScript with Web Thin Clients

In a Web Thin Client environment, the browser serves as the host for both HTML web pages published by the IWS Server, as well as the host for VBScript code segments that are associated with a particular Screen or object on the Screen. Generally, Microsoft Internet Explorer serves as the browser in a Web Thin Client environment. A InduSoft ActiveX Control (ISSymbol) is used to coordinate communications between the IWS Server and a Web Thin Client.

In a Windows XP/2000/NT-based Web Thin Client environment, Microsoft Internet Explorer (e.g. Version 6 or later) supports VBScripts and ActiveX by default. In a Windows CE-based Web Thin Client environment, Microsoft Internet Explorer (typically provided with PocketPC products) supports both VBScript and ActiveX, but VBScript support must be enabled in the Windows CE image (part of the Platform Build process, typically done by the hardware supplier). Windows CE systems with Microsoft Pocket Explorer (different that Microsoft Internet Explorer) will not work with VBScript as Pocket Explorer does not support VBScript due to memory limitations. Also remember that any ActiveX controls used on a Windows CE Web Thin Client must be developed to support Windows CE.

VBScript Interface	Functioning related to a Web Thin Client
Global Procedures	VBScript Global Procedures are accessible to VBScript code segments that execute on a Web Thin Client
Graphics Module	Operates on IWS Server PC only. Procedures and Variables not accessible to a Web Thin Client.
Screen Scripts	This VBScript interface (for a Web Page) executes independently from the VBScript Interface for a Screen running on the IWS Server. • The Graphics_OnStart() subroutine starts when the Web Thin Client Station is successfully logged in and ISSymbol is hosted on the Web Browser • The Graphics_WhleRunning() subroutine executes on the Web Thin Client while the Web Thin Client remains logged in and the ISSymbol Control remain hosted on the Web Browser • The Graphics_OnEnd() subroutine is executed once the Web Thin Client logs off or the ISSymbol Control is no longer hosted by the Web Browser
Command Dynamic	This VBScript interface (for a Web Page) executes independently from the VBScript Interface for a Screen running on the IWS Server.
ActiveX Event Handler	This VBScript interface (for a Web Page) executes independently from the VBScript Interface for a Screen running on the IWS Server.
Background Task Startup	Operates on IWS Server PC only. Procedures and Variables not accessible to a Web Thin Client.
Background Task Scripts	Operates on IWS Server PC only. Procedures and Variables not accessible to a Web Thin Client.

Key Notes:

- **Under Windows XP/2000/NT, to check or modify Internet Explorer's settings for support of VBScript and ActiveX Controls, open Internet Explorer, then click on Tools -> Internet Options -> Security -> Custom Level.**
- **All VBScript interfaces unique to the Web Thin Client continue to have access to IWS tags and IWS built-in functions.**
- **When using a Windows CE device for the Web Thin Client, be sure ActiveX support and VBScript support is enabled. This is a function of the Windows CE OS image built using Microsoft Platform Builder.**
- **When using a Windows CE device for the Web Thin Client, verify that MsgBox and InputBox functions are enabled in the Windows CE OS image if you intent to use them,**

VBScript Language Reference

This Language Reference section is intended to cover VBScript as it is intended to be used with InduSoft Web Studio (IWS) and CEView.

VBScript, or more properly Microsoft Visual Basic Scripting Edition, is one of the members of the Microsoft Visual Basic family. VBScript is primarily a subset of VBA, or Visual Basic for Applications, although VBA and VBScript are targeted at different applications. VBA was intended to be used for the automation of Microsoft Office and Microsoft Project applications, while VBScript was designed for use with Web-based applications, both on the client side (e.g. Microsoft Internet Explorer) where it compliments Jscript, and on the Server side, where it works with ASP (Active Server Pages) applications and WSH (Windows Script Host) scripts.

InduSoft provides a VBScript Hosting environment for the InduSoft Web Studio (IWS) and CEView HMI/SCADA software, allowing developers to use both VBScript programmability and native IWS (and CEView) configurability. This combination of development methodologies lets developers chose which development methodology best suits their application requirements. InduSoft has chosen to implement VBScript instead of VBA, since VBScript has a number of advantages not inherent in VBA, including the support for thin clients and Windows CE runtime environments.

This VBScript Language Reference covers the following material:

- Variables (Type, Declaration, Scope)
- Constants (Explicit, Implicit)
- Keywords
- Errors (Runtime, Syntax)
- Operators (Type, Precedence)
- Functions and Procedures
- Statements
- Objects and Collections
- VBScript restrictions within the IWS development environment
- Example VBScript Applications

VBScript Variables

Variable Data Types and Subtypes

VB and VBA are compiled languages that require you to explicitly declare the variables you are going to use and their data type. To explicitly declare a VB or VBA variable, you would use the **Dim** keyword. The following example shows how VB or VBA would declare the variable x as an integer:

```
Dim x As Integer
Dim a, b, c As Integer
```

With VBScript, you also use the **Dim** statement to explicitly declare a variable. However, you are not required to explicitly declare variables in VBScript. If you do not explicitly declare a variable, it is implicitly declared when the variable is used. However, typing (spelling) errors can typically go undetected until a problem occurs, and then must be detected and corrected. By adding the **Option Explicit** command at the beginning of the script, you can force the VBScript Scripting Engine to only use the variables that are explicitly declared.

```
Example    Dim a,b                  ‘ explicitly declares the variables a & b
           a = 4                    ‘ assigns the value of 4 to variable a
           b = 4                    ‘ assigns the value of 4 to variable b
           c = a + b                ‘ VBScript will create a variable c, and then perform the add
```

```
Example    Option Explicit          ‘ Force explicit definition of variables
           Dim a, b                 ‘ declare variables a and b
           a = 4                    ‘ define variable a
           b = 4                    ‘ define variable b
           c = a + b                ‘ will generate an error since c not explicitly declared
```

Interestingly, VBScript does not allow you declare the variable data type (i.e. integer, real, etc.) in the Dim statement. In fact, VBScript does not support data Type declarations. This is a major difference between VBScript and VB/VBA. Instead, all VBScript variables are of a data type called **Variant**, meaning the data type can be whatever is required. However, there are a variety of VBScript Variant data **subtypes** that correspond to traditional data types familiar to programmers. These variant data subtypes are:

Variant data subtypes

Subtype	Description
Array	An indexed list of variants
Boolean	Boolean value of either True or False. False has a value of 0, and True has a value of -1.
Byte	Contains integer in the range 0 to 255
Currency	Floating-point number in the range -922,337,203,685,477.5808 to 922,337,203,685,477.5807
Date(Time)	Contains a number that represents a date between January 1, 100 to December 31, 9999
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
Empty	Uninitialized Variant
Error	Contains an error number used with runtime errors
Integer	Contains integer in the range -32,768 to 32,767
Long	Contains integer in the range -2,147,483,648 to 2,147,483,647
Null	A variant containing no valid data
Object	Contains an object reference. Note that this is not the object itself.
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
String	Contains a variable-length string that can be up to approximately 2 billion characters in length.

This relationship between type **Variant** and the various data **subtypes** is explained by the fact that VBScript uses what is called “late-bound” typing, meaning that the data subtype is determined at runtime by its usage or by a function. At runtime, the Parser in the VBScript Scripting Engine will determine the data type required and allocate storage accordingly, then execute the statements or functions accordingly for the data type. Microsoft reportedly implemented VBScript with late-bound data typing in order to improve execution speed.

In its simplest form, a **Variant** contains either numeric or string data. A **Variant** behaves as a number when you use it in a numeric expression and as a string when you use it in a string expression. That is, if you are working with data that looks like numbers, VBScript assumes that it is a number 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. If you enclose a variable in quotation marks (" "), you will always make the variable behave as a strings. 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 to 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.**

Beyond simple numeric or string classifications, a **Variant** can be distinguished by the specific nature of data it contains or represents. For example, numeric information can represent date or time. When the variable is used with other date or time variables or operations, the result is always expressed as a date or a time.

The **Variant** type is best thought of as a super data type which includes all the data subtypes. You can change a variable's value and subtype at runtime by assigning a new value to the variable from one previously assigned to it. For example:

```
        Dim x
        x = "message1"                          ' x would be a string subtype
        x = 1                                   ' x would now become a integer subtype
```

Just to make this a little more confusing, if you had the statement

```
        x = 2.5
```

this could be a Currency, Single or Double data subtype. By default, VBScript would consider 2.5 to be a Double data subtype. In the previous example, the variable x which was assigned a value of 1 could be a data subtype of Boolean, Byte, Integer or Long. VBScript by default would consider the variable x with a value of 1 to be an Integer data subtype.

Key Notes:

- **The Dim keyword can be used to declare one or more variables. Multiple variables are separated by comma(s).**
- **Option Explicit requires all variables to be declared, and is helpful to reduce typing (spelling) errors**
- **The value assigned to a variable can be another variable, a named constant (implicit or explicit) or a literal. A literal is simply static data, and can be a string of text, number, date or a boolean value. E.g.**

```
        a = 2
        myStr = "Alarm 1 on"
```

Array Variables

Most variables discussed to this point have been of the type that contain a single value. These are called scalar variables. Variables that contain a series of values are called array variables. Scalar variables and array variables can be explicitly declared the same way using the **Dim** keyword, except that array variables use the parentheses () following the variable name, used to indicate the size of the array. An example of a single dimension array containing 10 elements is declared by:

```
Dim a(9)
```

All arrays in VBScript are zero-based, meaning that the number of array elements is always equal to the number of elements shown in the parentheses plus one. This is consistent with arrays in IWS.

Arrays that are declared with a number in the parentheses are called fixed-size arrays. Data can be assigned to each element of the array as follows:

```
Dim a(9)
a(0) = 1
a(1) = 20
a(2) = -3
```

Data can be retrieved from an element of the array by using an index into the array. For example:

```
Dim a(9), array_index, x, y
a(0) = 1
a(1) = 20
a(2) = -3
x = a(0)                              ‘ variable x is assigned a value of 1
array_index = 2
y = a(array_index)                    ‘ variable y is assigned a value of -3
```

Arrays can be multi-dimensional, with up to 60 dimensions. For a two-dimensional array, the first number is referred to as the number of rows, and the second number being the number of columns. Examples of multi-dimensional array declaration is as follows:

```
Dim a(4,9)                            ‘ array has 5 rows and 10 columns
Dim b(4,4,9)                          ‘ a 5 x 5 x 10 3-dimensional array
```

VBScript supports dynamic arrays, whose size can change during runtime. Dynamic arrays can change the number of dimensions and the size of any or all dimensions. These arrays are initially declared using the **Dim** (or **ReDim**) keyword followed by a closed parenthesis. Then, prior to using the dynamic array, the **ReDim** keyword is used to specify the number of dimensions and size of each dimension. The **ReDim** can subsequently be used to modify the dynamic array’s number of dimensions or size of each dimension. The **Preserve** keyword can be used to preserve the contents of the array as the resizing takes place. For example:

```
Dim MyArray(), x
ReDim MyArray(19)                     ‘ MyArray has 20 elements
MyArray(0) = 10                       ‘ Assign values to first 2 elements
MyArray(1) = 20
ReDim Preserve MyArray(24)            ‘ change MyArray to a 25 element array
x = MyArray(0)                        ‘ variable x is assigned value of 10
```

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

VBScript provides several functions for the manipulation of arrays. These include:

Array Functions & Statements

Array Functions	Description
Array	Returns a variant containing an array
Dim	Declares variables and allocates storage space
Erase	Reinitializes the elements of fixed-size arrays, deallocates dynamic-array storage space.
Filter	Returns a zero-based array that contains a subset of a string array based on a filter criteria
IsArray	Returns a Boolean value that indicates whether a specified variable is an array
Join	Returns a string that consists of a number of substrings in an array
LBound	Returns the smallest subscript for the indicated dimension of an array
ReDim	Declare dynamic array variables, allocates or reallocates storage space at procedural level
Split	Returns a zero-based, one-dimensional array that contains a specified number of substrings
UBound	Returns the largest subscript for the indicated dimension of an array

Examples using these array functions are:

```
Dim MyArray(3), MyString, VarArray(), MyIndex, littleArray
MyArray(0) = “President ”
MyArray(1) = “George ”
MyArray(2) = “W. ”
MyArray(3) = “Bush”
MyString = Join(MyArray)                  ‘ MyString equals “President George W. Bush”
MyString = “HelloxWidexWorld”
MyArray = Split (MyString, “x”, -1,1)     ‘ MyArray(0) contains “Hello”
                                          ‘ MyArray(1) contains “Wide”
                                          ‘ MyArray(2) contains “World”
MyIndex = Filter(MyArray, “W. “)          ‘ MyIndex will equal 2
ReDim VarArray(10)                        ‘ Redimension the VarArray array
ReDim VarArray(20)                        ‘ Redimension the VarArray array
VarArray(19) = 19
VarArray(20) = 20
littleArray = Array(12.3.64, 15)          ‘ Populate the array with the Array function
Erase VarArray                            ‘ Deallocates memory for the dynamic array
Erase MyArray                             ‘ Simply erases the fixed size array
```

Key Notes:

- **VBScript Array indices always start with 0. This is not the case with VBA.**
- **An array MyArray(2) has 3 elements, with indices 0, 1, and 2.**
- **Multi-dimensional arrays are supported up to 60 dimensions.**
- **Dim MyArray() is a dynamic array, and must be sized later with the ReDim statement .**
- **The Preserve keyword will preserve existing elements in a dynamic array**
- **Erase function deallocates memory for dynamic arrays, only clears fixed size arrays**

Boolean Variables

Boolean variables have one of two values; **True** or **False**. The VBScript Keywords **True** or **False** can be used to assign a value to the boolean variable. A boolean **False** is stored as a 0, but the boolean **True** is not stored as a 1. Since the data storage element for the boolean value is a signed 32-bit value, a boolean **True** will have all bits in the 32-bit value set to 1, which is a negative signed integer value of -1. It is best to work with the boolean values **True** or **False** when working with boolean variables.

Literal Keywords used with Boolean data subtypes

Keyword	Description
False	Boolean condition that is not correct (false has a value of 0)
True	Boolean condition that is correct (true has a value of -1)

An example would be:

```
Dim overtemp_condition
If $temperature > 100 then
    overtemp_condtion = True
Else
    Overtemp_condition = False
End If
```

There are several logical operators available in VBScript that can be used to evaluate logical expressions. These logical operators can be used with both Boolean data subtypes as well as in Comparison expressions. In the table below, a and b are assumed to represent logical expressions.

Logical Operators

Logic	Operator	Example	Returns
AND	And	a AND b	True only if a and b are both true
OR	Or, |	a OR b	True if a or b is true, or both are true
Exclusive OR	Xor	a Xor b	True if a or b is true, but not both
Equivalence	Eqv	a Eqv b	True if a and b are the same
Implication	Imp	a Imp b	False only if a is true and b is false otherwise true
NOT	Not	a Not b	True if a is false; False if a is true

A couple examples of the logical operators are:

```
Dim temp, pressure
If (temp > 212) And (pressure > 1) then       ‘ evaluate a conditional expression
    Call Alarm_routine
End If

Dim a, b, temp, pressure
a = (temp > 212) And (pressure > 1)           ‘ conditional expression stored as a boolean
If a = True Then                              ‘ logical condition test
    Call Alarm_routine
End If
```

Note that the **|** operator (shift \) can be used instead of the **Or** logical operator. The statements (a | b) and (a Or b) are equivalent.

Logical Truth Table

a	b	a And b	a Or b	a Xor b	a Eqv b	a Imp b	Not a
T	T	T	T	F	T	T	F
T	F	F	T	T	F	F	F
F	T	F	T	T	F	T	T
F	F	F	F	F	T	T	T

Byte, Integer & Long Variables

These three data subtypes are whole numbers that vary by the range of values that they can hold. Note that the Byte data subtype has only a positive range (i.e. it is an unsigned value), while Integer and Long are signed values. Byte is an 8-bit number, Integer a 16-bit number and Long a 32-bit number.

Subtype	Range
Byte	0 to 255
Integer	-32,768 to 32,767
Long	-2,147,483,648 to 2,147,483,647

There are several types of operations that can be performed on these data subtypes, such as arithmetic, comparison and logical operators. Also, many math functions can be used with these data subtypes. Some examples are:

```
Dim MyByte, MyInt, MyHex
MyByte = $input_val – 5                 ‘ read integer IWS tag input_val and subtract 5
If MyByte > 255 Then MyByte = 255       ‘ used in a condition statement. Make a byte value
MyInt = 459
MyHex = Hex(MyInt)                      ‘ returns 1CB
```

In addition to these functions, there are Byte versions of string operations that can be used with Byte data contained in strings. For example, data from a serial port might be stored in a string. Remember that strings can be essentially any length. The **Mid** function could be used to return a specified number of characters from a string, but the **MidB** function will return a specified number of Bytes from the string.

Currency Variables

VBScript supports a currency data type. The valid range for currency is from -922,337,203,685,477.5808 to 922,337,203,685,477.5807. You can perform most of the same operations on the currency data type as you can perform on other numbers. The primary difference is that the currency data subtype will contain the currency symbol, and is formatted using the FormatCurrency function.

Currency Format Function

Function	Description
FormatCurrency	Returns an expression formatted as a currency value

Example1:

```
Dim val, f_val                          ‘ This example limits the number of decimal places
val = 123.456                           ‘ assign a currency value to val
f_val = FormatCurrency(val, 2)          ‘ 2 digits after decimal, result is f_val = $123.45
```

Example2:

```
Dim price                               ‘ This example changes the currency symbol
price = 123.456
SetLocale(1033)                         ' Set locale to United States, use $ currency symbol
curDollars = FormatCurrency(price, 2)   ' curDollars set to $123.46
myLocale = SetLocale(2057)              ' Set locale to UK, use £ currency symbol
curPounds = FormatCurrency(price, 2)    ' curPounds set to £123.46
```

Note: To use the Euro € symbol for a country that uses the Euro, make sure the system’s Region Settings is properly set, otherwise the pre-Euro symbol will be used.

Date (and Time) Variables

Date is another of VBScript's data subtypes. The Date data subtype actually contains both date and time information that can be stored in variables and constants. The Date format is Gregorian and the Time is local, with Day Lights Savings changes ignored unless specified in the system settings. The date subtype is a number that represents a date in the range of January 1, 100 to December 31, 9999. The following are valid ranges for the date and time fields:

Second	0 to 59
Minute	0 to 59
Hour	0 to 23
Day	0 to 31
Month	1 to 12
Year	100 to 9999

With the date subtype, there are predefined VBScript constants that refer to the day of the week and New Year's week. There are also Date and Time formatting constants that are used with the FormatDateTime function. In addition, there are several Date and Time functions available in VBScript.

A literal date can be defined by surrounding a date/time value with the # symbol on each end.

Some examples using Date and Time include:

```
Dim CurDay, OldDay, DayDiff, HourDiff
Dim MyDay, MyMonth, MyYear, RecentDay, OtherDay, MyDate
OldDay = #3/27/2006 08:20:59#                ' Set an old date
CurDay = Now()                               ' reads current System time and date
DayDiff = DateDiff("d". OldDay. CurDay)      ' returns # days between OldDay and CurDay
HourDiff = DateDiff("h", OldDay, CurDay)     ' returns # hours between OldDay and CurDay
MyDay = 27                                   ' specify day, month, year
MyMonth = 3
MyYear = 2006
RecentDay = DateSerial(MyYear, MyMonth, MyDay)    ' converts into a Date subtype variable
OtherDay = DateSerial(MyYear, MyMonth-2, MyDay)   ' you can use expressions in this function
MyDate = FormatDateTime(CurDay, vbLongDate)       ' displays a date in the long format,
                                                  ' uses computer's regional settings
```

Days of Week Constants

Constant	Value	Description
vbUseSystem	0	Use system value
vbSunday	1	Sunday (Default)
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday

New Years Week Constants

Constant	Value	Description
vbUseSystem	0	Use system value
vbFirstJan1	1	Start with the week in which January 1st occurs (default)
vbFirstFourDays	2	Start with the week that has at least four days in the new year
vbFirstFullWeek	3	Start with the first complete week of the new year

Date and Time Format Constants (used with FormatDateTime function)

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.

Date and Time Functions

Function	Description
CDate	Converts a valid date and time expression to the variant of subtype Date
Date	Returns the current system date
DateAdd	Returns a date to which a specified time interval has been added
DateDiff	Returns the number of intervals between two dates
DatePart	Returns the specified part of a given date
DateSerial	Returns the date for a specified year, month, and day
DateValue	Returns a date
Day	Returns a number that represents the day of the month (between 1 and 31, inclusive)
FormatDateTime	Returns an expression formatted as a date or time
Hour	Returns a number that represents the hour of the day (between 0 and 23, inclusive)
IsDate	Returns a Boolean value that indicates if the evaluated expression can be converted to a date
Minute	Returns a number that represents the minute of the hour (between 0 and 59, inclusive)
Month	Returns a number that represents the month of the year (between 1 and 12, inclusive)
MonthName	Returns the name of a specified month
Now	Returns the current system date and time
Second	Returns a number that represents the second of the minute (between 0 and 59, inclusive)
Time	Returns the current system time
Timer	Returns the number of seconds since 12:00 AM
TimeSerial	Returns the time for a specific hour, minute, and second
TimeValue	Returns a time
Weekday	Returns a number that represents the day of the week (between 1 and 7, inclusive)
WeekdayName	Returns the weekday name of a specified day of the week
Year	Returns a number that represents the year

Key Notes:

- **VBScript Date and Time formats can change based on the user logged into the system. Reference Microsoft Knowledge Base Article 218964.** http://support.microsoft.com/kb/q218964/

- **VBScript Date and Time functions may not be formatted properly in non-English (US) locales. Reference Microsoft Knowledge Base Article 264063.** http://support.microsoft.com/default.aspx/kb/264063

Empty Variables

Empty is a single VBScript variable that has been declared, but has no explicitly assigned value. This is also known as an uninitialized variable. There are two ways a variable can be uninitialized. The first is when it is explicitly declared but has not yet been assigned a value. For example:

```
Dim a, b
a = 2                               ‘ a is initialized, b is still uninitialized
```

The second way a variable can be uninitialized is by assigning it a value of Empty. **Empty** is VBScript keyword. For example:

```
a = 2                               ‘ a is a integer variable
b = “Hello”                         ‘ b is a string variable
a = Empty                           ‘ makes variable a uninitialized
b = Empty                           ‘ makes variable b uninitialized
```

If the variable was a numeric data subtype and set to a value of Empty (making it a Empty subtype), its value will be 0. If the variable was a string data subtype and set to a value of Empty, its value will be “”. The numeric and string subtypes can still be used in statements without generating a VBScript error although their values were set to Null

Note that a variable being **Empty** is different that variable having a **Null** value. An Empty variable is uninitialized, while a Null variable contains no valid data.

Error Variables

A variable with an Error data subtype contains an error number generated by the VBScript Parser or Runtime Engine (signifying the VBScript Syntax error or Runtime error). An Error variant data subtype can only be created by the VBScript Parser or Runtime Engine, or by calls to VBScript Object Methods. The programmer cannot directly create or manipulate Error data subtypes.

See the **Err** Object for examples of how to use errors.

Null Variables

A Null variable is a single variable that indicates the variable contains no valid data. A null value is typically used to represent missing data. A variable becomes a Null variable when it is assigned a null value by using the **Null** keyword. For example:

```
Dim a,b
a = 2                               ‘ a is initialized, b is uninitialized
a = Null                            ‘ a is Null, b is uninitialized (Empty)
```

One of the main differences between Empty and Null is that a variable can be of type Empty (uninitialized) when it is declared but not assigned a value, or when it is assigned a value of Empty. A Null variable, on the other hand, must be assigned a Null value.

Object Variables

The Object data subtype references an object. Although the topic of objects will be covered in detail later, at this point it is worth noting that there are two types of Objects; Intrinsic (i.e. VBScript-based) and Extrinsic Objects.

Intrinsic Objects are pre-defined by VBScript. VBScript includes the intrinsic **Err** object for error handling. The programmer can only use the **Err** object name for this object.

Extrinsic objects need to be declared, or instantiated (i.e. create an instance of the Object). With extrinsic objects, the programmer defines an object name in the declaration statement. The object name can be any valid variable name allowed by VBScript, although following variable naming conventions is strongly suggested.

Depending on the type of extrinsic object, different statements are used to instantiate the object. For example, with user-defined Classes, you would use the following format to instantiate the object.

```
Set cObj = New classname
```

where cObj is the name of the new object being instantiated, **New** is a VBScript Keyword, and classname is the name of the user-defined class, which is merely a template for the object.

Other extrinsic objects include ActiveX Objects, ADO.NET, and OLE Automation Objects such as Microsoft Office applications and components. These objects use a different statement format for instantiation. They use either the **CreateObject** or **GetObject** functions. For example:

```
Set cObj = CreateObject("ADODB.Connection")
Set xlObj = CreateObject("Excel.Application")
Set xlBook = GetObject("C:\Test.XLS")
```

The difference between CreateObject and GetObject is that CreateObject is used to create an interface to a new instance of an application (or object) while the GetObject is used with an application that is already loaded.

Real (Single, Double) Variables

Real data types in VBScript are floating point numbers that can be either single precision (**Single**) or double precision (**Double**). Their ranges are:

Single	-3.402823E+38 to -1.401298E-45 for negative values
	1.401298E-45 to 3.402823E+38 for positive values
Double	-1.79769313486232E+308 to -4.94065645841247E-324 for negative values
	4.94065645841247E-324 to 1.79769313486232E+308 for positive values

There are several types of operations that can be performed on the Real data subtype, such as arithmetic, comparison and logical operators. Also, many math functions can be used with this data subtypes. Some examples are:

```
Dim R1, R2, R3, Radius
R1 = 3.14159
Radius = 2
R2 = R1 * radius * radius
R3 = FormatNumber (R2, 2)                ' R3 equals 12.57 (R2 = 12.56636)
```

Number Format Functions

Function	Description
FormatNumber	Returns an expression formatted as a number
FormatPercent	Returns an expression formatted as a percentage

Key Notes:

1. **If you use a floating point number in VBScript, you are better off to use a Double. When VBScript performs all floating point operation on a Single, VBScript will first convert the Single into a Double, performs the operation, and converts back to a Single. Additionally, VBScript Single is not smaller in memory size than a VBScript Double.**

Strings Variables

VBScript supports the String data subtype. Strings are variable length, limited only by the available system memory. In practice of course, they are not that long. Strings are a set of characters enclosed in (double) quotation marks. Variables are assigned a string value in the following manner:

```
Dim str
str = “hello”
```

The quotation marks signify the beginning and the end of a string. If you want to embed quotation marks in a string (without intending to signify the end of the string), you need to use two double quotation marks (adjacent) to embedded one of the quotation marks. For example,

```
Dim msg
msg = “Mr. Smith says “”hello”””                  ‘ String data is: Mr. Smith says “hello”
```

VBScript has predefined string constants that can be used for formatting strings used for text messages. These string constants include:

String Constants

Constant	Value	Description
vbCr	Chr(13)	Carriage return
vbCrLf	Chr(13) & Chr(10)	Carriage return and linefeed combination
vbFormFeed	Chr(12)	Form feed
vbLf	Chr(10)	Line feed
vbNewLine	Chr(13) & Chr(10) or Chr(10)	Platform-specific newline character
vbNullChar	Chr(0)	Null Character
vbNullString	Null String	Null String - Not the same as a zero-length string ("")
vbTab	Chr(9)	Horizontal tab
vbVerticalTab	Chr(11)	Vertical tab

Strings can be easily concatenated by use of the **&** operator. For example:

```
Dim str
str = “hello”
str = str & “ world”                    ‘ variable str now contains the string “hello world”
```

Using the string concatenation operator, another method of adding embedded quotation marks (or other characters) to a string would be:

```
Dim str, str_quotemark
str_quotemark = chr(34)
str = “Mr. Smith says” & str_quotemark & “hello” & str_quotemark
```

While VBScript string handling capability can be very useful, programmers should be aware of information given in Microsoft Knowledge Base Article 170964[1]. This article states that when strings get

[1] See http://support.microsoft.com/kb/q170964/

very large (e.g. 50kB or larger), the time to concatenate these strings can be very long. For example, a typical string concatenation where:

```
        Dim dest, source                          ‘ String variables
        Dim i, N
        For i = 1 to N
            dest = dest & source
        Next N
```

Using the programming method above, the Article notes that the length of time to perform the concatenation increase proportionately to N-squared. This increase in time is due to the method VBScript uses to concatenate strings, which is:

- allocate temporary memory large enough to hold the result.
- copy the dest string to the start of the temporary area.
- copy the source string to the end of the temporary area.
- de-allocate the old copy of dest.
- allocate memory for dest large enough to hold the result.
- copy the temporary data to dest.

The Article details a method using the **Mid$** statement and pre-allocation of memory to significantly reduce the time to concatenate large strings. Also, you can reference the section on Classes for another method to speed string concatenation.

There are several functions available to manipulate strings. Refer to the reference material in the Appendix for a detail description of these functions.

String Functions

Function	Description
InStr	Returns the position of the first occurrence of one string within another. The search begins at the first character of the string
InStrRev	Returns the position of the first occurrence of one string within another. The search begins at the last character of the string
LCase	Converts a specified string to lowercase
Left	Returns a specified number of characters from the left side of a string
Len	Returns the number of characters in a string
LTrim	Removes spaces on the left side of a string
Mid	Returns a specified number of characters from a string
Replace	Replaces a specified part of a string with another string a specified number of times
Right	Returns a specified number of characters from the right side of a string
RTrim	Removes spaces on the right side of a string
Space	Returns a string that consists of a specified number of spaces
StrComp	Compares two strings and returns a value that represents the result of the comparison
String	Returns a string that contains a repeating character of a specified length
StrReverse	Reverses a string
Trim	Removes spaces on both the left and the right side of a string
UCase	Converts a specified string to uppercase

Data Subtype Identification

The Parser that is part of the VBScript Scripting Engine automatically defines a variable's data subtype for you at runtime. However, there are times when the programmer may need to know the variable's data subtype. To determine the specific data subtype used, VBScript you can use any of the three categories of functions to determine the data subtype:

- The **VarType**(*variable*) function which returns a code based on the **Variant** data subtype used
- Various **IsXxxx**(*variable*) functions which return boolean values indicating whether the variable is of a specific data subtype.
- A **TypeName**(*variable*) function which returns a string based indicating the data subtype

Variant Data Subtype Identification Functions

Variant Function	Description
IsArray()	Returns a Boolean value indicating whether a variable is an array
IsDate()	Returns a Boolean value indicating whether an expression can be converted to a date
IsEmpty()	Returns a Boolean value indicating whether a variable has been initialized.
IsNull()	Returns a Boolean value that indicates whether an expression contains no valid data (Null).
IsNumeric()	Returns a Boolean value indicating whether an expression can be evaluated as a number
IsObject()	Returns a Boolean value indicating whether an expression refers to a valid Automation object.
TypeName()	Returns a string that provides Variant subtype information about a variable
VarType()	Returns a value indicating the subtype of a variable

VarType() Function

This function is similar to **TypeName** except that a numeric value, or ID, is returned that is used to identify the data subtype. This ID can then, as an example, be used in a flow control statement.

VarType Constants (returned from the VarType() function)

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	Object
vbError	10	Error
vbBoolean	11	Boolean
vbVariant	12	Variant (Used only with Arrays)
vbDataObject	13	Data-access Object
vbDecimal	14	Decimal
vbByte	17	Byte
vbArray	8192	Array

Example:

```
Myval = 23.3
If VarType(Myval) = vbSingle Then
    Msgbox "MyVal is a Single Precision Floating Point Number"
End If
```

Key Notes:

2. **If VBScript detects that the variant subtype is an Array, the VarType function never returns 8192. This is a base figure, and the actual value returned is 8192 plus the value of the array type. For instance, if you have an Array of Boolean values, the value returned would be vbArray + vbBoolean, or 8192 + 11 = 8203.**

IsXxxx() Functions

This is a series of functions that lets you determine whether a specific variable or constant is a certain data subtype. These functions check the variable or constant against a specific data subtype and return a Boolean value (**True** or **False**) indicating whether the variable or constant is the specified data subtype. Examples include:

```
Dim MyArray(5)                    ‘ Declare an array
Dim MyVal                         ‘ Declare a variable
Date1 = “April 14, 2006”          ‘ Assign Date
Date2 = #6/10/89#                 ‘ Assign Date
Date3 = “Hello World”             ‘ Assign string
MyCheck = IsArray(MyArray)        ‘ Returns a Boolean True
MyCheck = IsDate(Date1)           ‘ Returns a Boolean True
MyCheck = IsDate(Date2)           ‘ Returns a Boolean True
MyCheck = IsDate(Date3)           ‘ Returns a Boolean False
MyCheck = IsEmpty(MyVal)          ‘ Returns a Boolean True
MyVal = 5                         ‘ Assign a value of 5
MyCheck = IsNumeric(MyVal)        ‘ Returns a Boolean True
MyCheck = IsEmpty(MyVal)          ‘ Returns a Boolean False
MyCheck = IsNull(MyVal)           ‘ Returns a Boolean False
MyVal = Null                      ‘ Assign a null value (contains no valid data)
MyCheck = IsNull(MyVal)           ‘ Returns a Boolean True
MyVal = Empty                     ‘ Assign Empty (uninitialized state)
MyCheck = IsEmpty(MyVal)          ‘ Returns a Boolean True
```

Alternatively, you can use the **IsXxxx()** function in a conditional statement. For example,

```
Dim sInput
sInput = InputBox (“Enter a data value”)
If IsNumeric (sInput) Then
    MsgBox “Valid Input”
Else
    Msgbox “Invalid Input”
EndIf
```

TypeName() Function

TypeName is a read-only function that identifies the data subtype and returns a string that contains the data subtype. This string can then be used in a flow control statement, or in a message.

Return values from TypeName function

Return Value	Description
<object type>	Actual Type name of an Object
Boolean	Boolean value (**True** or **False**)
Byte	Byte value
Currency	Currency value
Date	Date or Time value
Decimal	Decimal value
Double	Double-precision floating-point value
Empty	Uninitialized
Error	Error
Integer	Integer value
Long	Long integer value
Nothing	Object variable that doesn't yet refer to an object instance
Null	No valid data
Object	Generic object
Single	Single-precision floating-point value
String	Character string value
Variant()	Variant Array
Unknown	Unknown object type

```
Dim MyVal
Dim a(9)
MsgBox TypeName(MyVal)              ' Will get message "Empty"
MyVal = 5.2
MsgBox TypeName(MyVal)              ' Will get message "Double"
Msgbox Typename(a)                  ' Will get message Variant()"
```

Key Notes:

3. **When you pass an Array argument to the TypeName function, it will return value of Variant(). This return value is not listed in Microsoft's official documentation. Since VBScript does not support data typing, there is no way to determine the data type of the array. Instead, you must determine the data type of each element in the array, one element at a time.**

Data Subtype Conversion

VBScript provides several functions that convert a VBScript variable from one data subtype to another. Since VBScript uses the **Variant** data type, these functions are not generally required. However, when passing data between IWS (or CEView) and VBScipt, or calling built-in IWS functions from VBScript where variables need to be put into the proper argument format, these VBScript data subtype conversion functions can be very useful.

Data Subtype Conversion Functions

Function	Description
CBool()	Converts an expression to a variant of subtype Boolean
CByte()	Converts an expression to a variant of subtype Byte
CCur()	Converts an expression to a variant of subtype Currency
CDate()	Converts a valid date and time expression to the variant of subtype Date
CDbl()	Converts an expression to a variant of subtype Double
CInt()	Converts an expression to a variant of subtype Integer
CLng()	Converts an expression to a variant of subtype Long
CSng()	Converts an expression to a variant of subtype Single
CStr()	Converts an expression to a variant of subtype String

Example:

```
a = -5.2
b = 4
c = “A”
LState = True
StartDate = #4/6/2005#
StartTime = #10:05:20#
d = CByte(a)                    ‘ Will generate overflow error, Bytes are only positive
d = CByte(d)                    ‘ d will equal 5 (Byte)
d = CStr(a + b)                 ‘ d will be “-1.2” (string)
d = CDate(StartDate + 20)       ‘ d will be 4/26/2005 (date)
d = CDate(StartTime)            ‘ d will be 10:05:20 am (date/time)
d = CDate(StartTime + 20)       ‘ d will be 1/19/1900 10:05:20 am (date/time)
d = CDate(StartTime + #1:5#)    ‘ d will be 11:10:20 am
d = CInt(LState)                ‘ d will be -1
```

Since VBScript does not use explicit data typing, one might expect that you would not get a type mismatch error. This however, is not necessarily true. For example, if you attempted to sum a number and a string, you will get a type mismatch error. Users are not allowed to freely mix heterogenous data even if all data is of type **Variant**. Again, type **Variant** allows the variable and constant data type to be determined at runtime, instead of being explicitly predefined.

More detail on the Data Subtype Conversion Functions is provided in the VBScript Functions section.

Key Notes:

- **You can’t pass an alphanumeric string to a conversion function that returns a number (e.g. CInt() or CLng() functions) if the string has more than one character containing an ASCII number. If you try to do this, a type mismatch error will occur**
- **The CStr() function provides the greatest flexibility when converting an expression into a String data subtype. If you use the CStr() function with a Boolean expression, the result will be a string of either “True” or “False”. If you use the CStr() function with a date expression, the date string will follow the operating systems short date format.**
- **To convert a string into a date data subtype, you can use either the CDate() function, or simply assign a date value to a variable, enclosing the value in hashes (#) as follows:**

  MyDate = “#3/22/2006#”

VBScript Naming Rules and Conventions

VBScript has some simple standard rules that apply to all VBScript variable names. These are:

- Must begin with an alpha character (A...Z)
- After the first character, they can contain letters, digits and underscores. No other embedded characters are permissible.
- Must be less than 255 characters in length
- Must be unique in the scope in which they are declared
- Cannot use names that are Keywords

Permissible	Not permissible
a	class.item
b2	+a
c_34_	@Test123

Microsoft recommends following a naming convention for variables, based on their data type. The variable name would contain a prefix, signifying its data type. Microsoft used the **vb** prefix for VBScript defined constants, and it is recommended to avoid using these prefixes with variables. The Microsoft recommended prefixes for programmer defined variables and constants are:

Microsoft Suggested Naming Convention for Variables

Data Subtype	Prefix	Example
Boolean	bln	blnFound
Byte	byt	bytRasterData
Currency	cur	curTotal
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
Variant	Var	varNumber

The Microsoft suggested naming convention are part of the "Hungarian Notation Standard" prefixes, developed by Microsoft in 1972. Although many of the other prefixes are for C++ programmers, there are a couple other Hungarian Notation prefixes that might be useful:

Additional Hungarian Notation Prefixes

Use	Prefix	Example
Pointer	p	pIndex
Class	c	cObject
Float	f	fCalc
Nested Class	X	X

Key Notes:

- **Good programming would suggest that variable names are descriptive**
- **While VBScript variable names are not case sensitive, the name of an ActiveX control inserted by IWS is case sensitive when referenced from a VBScript code segment.**
- **Most VBScript naming rules can be overridden by enclosing the name in brackets. For example, [@a.1] would be a valid VBScript name.**

Variable Scope

All VBScript variables have “scope”. Scope defines a variable’s visibility or accessibility from one procedure (or VBScript Interface) to another, which in IWS is principally determined by where you declare the variable. As a general rule, when you declare a variable within a procedure, only code within that procedure can access or change the value of that variable. This is called local scope and is for 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. However, as previously noted, InduSoft enforces certain restrictions on the scope of Variables and Procedures.

A variable’s lifetime depends on how long it exists (i.e. for how long memory is allocated for the variable). The lifetime of a script-level variable extends from the time it is declared until the time the script is finished running, or until the memory is released (e.g. **SET** obj **= Nothing** statement).

At procedure level, a variable exists only for as long as you are in the procedure. When the procedure exits, the variable is destroyed, and the memory previously allocated to the variable is released. Local variables are ideal as temporary storage space when a procedure is executing. Local variables with the same name can exist in several different procedures since the local variable is recognized only by the procedure in which it is declared.

VBScript allows for explicit declaration of the scope of a variable through the **Public** or **Private** declarations. These declarations can also define the size of an array. The **Public** or **Private** declarations must be made at the beginning of a script, while the **Dim** declaration can be made at any point in the script. When using the **Public** or **Private** declarations with IWS, be sure to use them in the variable declaration section. You cannot use **Public** or **Private** declarations in IWS with Global Procedures, Command Dynamic or ActiveX events (these are sections in IWS where VBScript can be placed). Note that the use of the **Public** declaration of a variable may be limited by IWS, as **Public** variables defined in one section in an IWS application are not necessarily accessible in another section. See the *VBScript Configuration and Operation in IWS* section for more details on this topic.

Example:
```
Sub MySub(a,b)
    Dim c
    c =a + b
End Sub

Call MySub (1,2)                 ‘ Call the subroutine MySub
MsgBox c                         ‘ c will be uninitialized, not the same variable as in
```

Example:
```
Sub Calc
    Dim a
    a = 6
End Sub

Dim a
a = 2
GoSub Calc
MsgBox “a = ” & a                ‘ a would equal 2, not 6
```

Example:
```
Private MyArray(5)               ‘ Private variables
Public MyVal, MyList(5)          ‘ Public variables
```

The following table is a brief summary of a VBScript variable's scope based on which IWS module the variable is declared in.

VBScript Variable Scope based on IWS module

IWS Module	Scope of Variables
Global Procedures	Variables accessible only within Global Procedures
Graphics Scripts	Variables accessible only within Graphics Script interface
Screen Scripts	Variables accessible only within the Screen where the Script is written
Command Dynamic	Variables accessible only in the IWS object where the Script is configured
ActiveX Events	Variables accessible only in ActiveX object where the Script is configured
Background Startup Script	Variables accessible within Background Startup Script and all Background Script Groups, but no where else
Background Script Groups	Variables accessible only within the Background Script Group where it is declared

Key Notes:

- **A variable's scope is determined by where the variable is located (i.e. in a Subroutine or Procedure, or in a main code segment)**
- **A variable's scope can be made Public or Private via Public and Private statements. These statements can also be used to declare the variable (allocate storage).**
- **With IWS, be sure to use the Public and Private declarations in the variable declaration section. You cannot use the Public or Private declarations in Global Procedures, Command Dynamic, or ActiveX events (see below).**
- **IWS places further limits on a variable's scope. Using the Public statement does not insure the variable is accessible by all VBScript code segments.**

VBScript Constants

VBScript supports both *explicit* and *implicit* constants. Constants should never be used as variables, and you can only assign a fixed value to a constant; assigning a variable to a constant is not allowed.

Explicit constants are defined by the programmer. Explicit constants have a defined value which, unlike a variable, is not allowed to change during the life of the script.

Implicit constants are pre-defined by VBScript. VBScript implicit constants usually begin with a **vb** prefix. VBScript implicit constants are available to the VBScript programmer without having to define them. Other objects, such as those used by ADO.NET, also have implicit constants predefined, usually with different prefixes. However, the implicit constants for these objects may not be know to VBScript and if not, will have to be defined as an explicit constant.

Constants have scope similar to variables. Implicit constants have scope throughout a VBScript program, while explicit constants can have the same or a more limited scope. You can use the **Private** or **Public** keyword in front of the **Const** declaration statement to define the scope of the constant. Keep in mind that the scope of a constant be have further limitations placed on it by IWS. Constants declared at the script level (or code segment level) have scope within the script, whether used in the code, procedures, functions or user-defined classes. Constants declared inside of a procedure or function have procedure-level scope, and cannot be used outside of the procedure or function.

Key Notes:

- **Use named constants instead of literals, especially if a literal is used more than once. This will help reduce programming errors, and allow changes to be made from one location. E.g.**

```
Const max_speed = 200                  ← Preferred method using constant
Dim speed
If speed >= max_speed Then GoSub SlowDown

vs.

Dim speed                              ← Non-preferred method using literal
If speed >= 200 Then GoSub SlowDown
```

- **Use the same naming rules for constants as for variables. Some authors recommend using all capital letters for constants to easily differentiate them from variables.**

Explicit Constants

An explicit constant is one which has an explicitly defined value, such as a number, string or other data subtype, assigned to a name by the programmer. The constant cannot be changed during the lifetime of the script. Constants are used in place of explicit values, making the VBScript easier to read and allowing for changes to be made simply.

The constant name needs to follow the same rules as VBScript variable naming. Some authors advocate using all capital letters for constants in an effort to distinguish them from variables.

To create an explicit constant, you use the keyword **Const**. You cannot use a function or another constant as part of the explicit value. You cannot use an expression with a VBScript Operator. For example:

```
Const Threshold = 101.5          ‘ Explicit constant Threshold has a value of 101.5
Const MyColor = &hFFFF           ‘ assigns a color constant to MyColor
Const CrLf = Chr(13) & Chr(10)          ‘ Not allowed to use a function
Const MyVal = 2 + 4                     ‘ Not allowed to have an operator in assignment
```

String literals are enclosed in double quotation marks (“), while date and time literals are enclosed in hashes (#). For example:

```
Const  MyString = “Hello World”
Const  StartDate = #4-1-2006#
```

After creating the constant, you can use the constant name in lieu of specifying an explicit value. For example:

```
Dim Alarm1, Alarm2
Const Threshold = 101.5                 ‘ Create a constant, value = 101.5
Alarm1 = Threshold                      ‘ Assigns the constant to the variable Alarm1
Alarm2 = Threshold + 5                  ‘ Adds 5 to the constant and assigns to Alarm2
```

Key Notes:

- **Use the following formats to assign constant values**

```
Const MyVal = &hFFFF                to assign a hexadecimal value
Const MyVal = "Hello"               to assign a string value
Const MyVal = “He was “”lost”””     to embed quotation marks
Const MyVal = #9-11-2001#           to assign a date and time literal
```

- **Use the same naming rules for variables as for constants**
- **Can’t use functions or operators in the assignment statement**

Implicit Constants

VBScript defines a number of implicit (or *intrinsic*) constants that can be used by the programmer in VBScript code, regardless of the location of the VBScript code in the IWS development environment. The intrinsic constants are grouped into various categories based on their use. For example, Color Constants are used to define a color, instead of entering a hex value. VBScript defines the following different categories of intrinsic Constants:

- Color Constants
- Comparison Constants
- Date and Time Constants
- Date Format Constants
- Days of Week Constants
- New Years Week Constants
- Error Constants
- Errors - VBScript Runtime
- Errors - VBScript Syntax
- File Attribute
- File Input/Output
- MsgBox Constants (determines what buttons appear and which are default)
- MsgBox Function Constants (identifies what buttons have been selected)
- SpecialFolder Constants
- String Constants
- Tristate Constants
- VarType Constants
- Locale ID (LCID)

VBScript implicit constants do not need to be defined by the programmer, they are predefined by VBScript. The VBScript implicit constants start with a prefix of “vb”. Some examples are:

```
    vbBlack                     ‘ The implicit color constant for black
    vbFriday                    ‘ The implicit day of week constant for Friday
    vbCrLf                      ‘ Implicit string constant for a Cr and a Lf
```

The following is a list of the various VBScript implicit constants:

Color Constants

Constant	Hex Value	Decimal Value	Description
vbBlack	&h00	0	Black
vbRed	&hFF	255	Red
vbGreen	&hFF00	65280	Green
vbYellow	&hFFFF	65535	Yellow
vbBlue	&hFF0000	16,711,680	Blue
vbMagenta	&hFF00FF	16,711,935	Magenta
vbCyan	&hFFFF00	16,776,960	Cyan
vbWhite	&hFFFFFF	16,777,215	White

Comparison Constants

Constant	Value	Description
vbBinaryCompare	0	Binary Comparison
vbTextCompare	1	Text-based Comparison

VBScript Date and Time Format Constants

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.

VBScript Days of Week Constants

Constant	Value	Description
vbUseSystem	0	Use system value
vbSunday	1	Sunday (Default)
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

VBScript New Years Week Definition

Constant	Value	Description
vbUseSystemDayOfWeek	0	Use system value for the first day of the week.
vbFirstJan1	1	Start with the week in which January 1st occurs (default). This is the default value for both DateDiff and DatePart.
vbFirstFourDays	2	Start with the week that has at least four days in the new year
vbFirstFullWeek	3	Start with the first complete week of the new year

VBScript Error Constants

Constant	Value	Description
vbObjectError	-2147221504	The base error number, to which a user-defined error number is added when a user-defined error is raised. To raise error #1000, the following format should be used: Err.Raise Number = vbObjectError + 1000

VBScript Runtime Errors

Error Number	Description
5	Invalid procedure call or argument
6	Overflow
7	Out of Memory
9	Subscript out of range
10	This array is fixed or temporarily locked
11	Division by zero
13	Type mismatch
14	Out of string space
17	Can't perform requested operation
28	Out of stack space
35	Sub or function not defined
48	Error in loading DLL
51	Internal error
91	Object variable not set
92	For loop not initialized
94	Invalid use of Null
424	Object required
429	ActiveX component can't create object
430	Class doesn't support Automation
432	File name or class name not found during Automation operation
438	Object doesn't support this property or method
445	Object doesn't support this action
447	Object doesn't support current locale setting
448	Named argument not found
449	Argument not optional
450	Wrong number of arguments or invalid property assignment
451	Object not a collection
458	Variable uses an Automation type not supported in VBScript
462	The remote server machine does not exist or is unavailable
481	Invalid picture
500	Variable is undefined
502	Object not safe for scripting
503	Object not safe for initializing
504	Object not safe for creating
505	Invalid or unqualified reference
506	Class not defined
507	An exception occurred
5008	Illegal assignment
5017	Syntax error in regular expression
5018	Unexpected quantifier
5019	Expected ']' in regular expression
5020	Expected ')' in regular expression
5021	Invalid range in character set

VBScript Syntax Errors

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

VBScript MsgBox Constants (settings)

Constant	Value	Description
vbOKOnly	0	Display **OK** button only. This is the default value
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** (question mark) 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.
vbMsgBoxRight	524288	Right align text
vbMsgBoxRtlReading	1048576	On Hebrew and Arabic systems, specifies that text should appear from right to left.
vbMsgBoxSetForeground	65536	Makes the message box the foreground window
vbApplicationModal	0	Application modal. The focus cannot move to another interface in the application until the dialog is closed
vbSystemModal	4096	System modal. On Win32 systems, this constant provides an application modal message box that always remains on top of any other programs you may have running.
vbMsgBoxHelpButton	16384	Help button.

VBScript MsgBox Function Constants (identifies what buttons have been selected)

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.

VBScript String Constants

Constant	Value	Description
vbCr	Chr(13)	Carriage return
vbCrLf	Chr(13) & Chr(10)	Carriage return and linefeed combination
vbFormFeed	Chr(12)	Form feed
vbLf	Chr(10)	Line feed
vbNewLine	Chr(13) & Chr(10) or Chr(10)	Platform-specific newline character
vbNullChar	Chr(0)	Null Character
vbNullString	0	Null String - Not the same as a zero-length string ("")
vbTab	Chr(9)	Horizontal tab
vbVerticalTab	Chr(11)	Vertical tab

VBScript Tristate Constants

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

The tristate constants are used when there are three possible options: True, False and Default.

VBScript VarType Constants (defines the possible subtypes of variables)

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	Object
vbError	10	Error
vbBoolean	11	Boolean
vbVariant	12	Variant (Used only with Arrays)
vbDataObject	13	Data-access Object
vbByte	17	Byte
vbArray	8192	Array

These constants are used with the VarType() function.

VBScript Locale ID (LCID) Chart

Locale	Short String	Hex Value	Dec. Value	Locale	Short String	Hex Value	Dec. 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	Macedonian (FYROM)	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 - Bokml	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 SAR	zh-hk	0x0C04	3076	Setsuana	tn	0x0432	1074
Chinese - Macau SAR	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 - 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 - Caribbean	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 – UK	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	Southern Sotho	st	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

Locale	Short String	Hex Value	Dec. Value	Locale	Short String	Hex Value	Dec. Value
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				

Common VBScript Locale ID (LCID) Chart (partial list)

Locale	Short String	Hex Value	Dec. Value
English - United States	en-us	0x0409	1033
English – UK	en-gb	0x0809	2057
German - Germany	de-de	0x0407	1031
Spanish - Mexico	es-mx	0x080A	2058
Chinese - China	zh-cn	0x0804	2052
Japanese	ja	0x0411	1041
French - France	fr-fr	0x040C	1036
Russian	ru	0x0419	1049
Italian - Italy	it-it	0x0410	1040

Key Notes:

- **You cannot re-assign a value to an implicit VBScript Constant. E.g.**

```
vbNull = 5                    ' Will generate an error. vbNull = 1
```

- **Use implicit constants instead of literals where possible in order to improve code readability.**

Declaring Variables, Objects and Constants

VBScript does not require the explicit declaration of scalar variables, i.e. those variables with only one value assigned at any given time. Arrays, Objects (except **Err**) and Constants must be declared. Scalar variables used but not declared are called implicit variables. While it may initially be convenient not to declare variables, any typing (spelling) errors of the variable or constant names may produce unexpected results at runtime.

The **Option Explicit** statement can be invoked at the beginning of the variable declaration script segment to force the declaration of variables. This statement must be placed above the first **Dim** statement. Any variables not declared will invoke an error message "Variable is undefined".

All variables and constants must follow the variable naming rules, and should follow standard naming conventions although not required to do so. Multiple assignments can be made on the same line when the variable declarations are separated by the colon character :.

Example:

```
Dim a, b, c                  ' Declares variables a, b & c
Dim k(9)                     ' Declares an array k with 10 elements
                             '  since VBScript is 0 based
a = 2 : b = 3 : c = 4        ' Assign values to variables
d = a + b + c                ' Implicitly defined variable d, equals 9
```

Example:

```
Option Explicit              ' Force explicit variable declaration
Dim a, b, c                  ' Declares variables a, b & c
a = 2 : b = 3 : c = 4        ' Assign values to variables
d = a + b + c                ' Error since d not explicitly defined
```

Scalar variables and Fixed-sized Arrays are declared using the **Dim** statement. Fixed-sized arrays have a defined number of dimensions and defined size to each dimension that do not change during the life of the variable.

Dynamic arrays are a type of array that can be dynamically resized during runtime. Dynamic arrays are initially declared using the **Dim** statement followed by closed parentheses. Then, at one or more points in the program, the **ReDim** statement is used to dynamically resize the array. For example:

```
Dim myDynamicArray()         ' Declare a dynamic array
ReDim myDynamicArray(10)     ' Now declare it to have 11 elements
```

The **Array** function can be used to initially populate the array. The following is an example

```
Dim myArray
myArray = Array(4.56.82. 3.82)
```

Extrinsic Objects must also be declared. Depending on the type of extrinsic object, different statements are used to instantiate (declare and allocate memory for) the object. For example, with user-defined Classes, you would use the following format to instantiate the object.

```
Set cObj = New classname
```

where cObj is the name of the new object being instantiated, **New** is a VBScript Keyword, and classname is the name of the user-defined class, which is merely a template for the object.

Other extrinsic objects include ActiveX Objects, ADO.NET, and OLE Automation Objects such as Microsoft Office applications and components. These objects use a different statement format for instantiation. They use either the **CreateObject** or **GetObject** functions. For example:

```
Set cObj = CreateObject("ADODB.Connection")
Set xlObj = CreateObject("Excel.Application")
Set xlBook = GetObject("C:\Test.XLS")
```

The difference between CreateObject and GetObject is that CreateObject is used to create an interface to a new instance of an application (or object) while the GetObject is used with an application that is already loaded.

To declare constants, you use the **Const** statement. An example is:

Const mySetting = 100

As previously discussed, constants have scope. The scope of a constant can be modified by adding either the keyword **Public** or **Private** in front of the Const declaration.

Key Notes:

- **All Arrays in VBScript are zero-based, meaning that the array myArray(10) really has 11 elements. Unlike VB or VBA, all arrays in VBScript are zero-based.**
- **Arrays, Objects (except implicit Err Object) and Named Constants must be declared.**
- **Using Option Explicit forces all variables to be declared. This helps prevent runtime errors due to mis-typing.**

VBScript Keywords

VBScript has many keywords. These keywords include the built-in constants and literals, operators, functions, statements and objects. These keywords are reserved, i.e. they cannot be used as names of variables or constants.

We have already covered the VBScript built-in implicit constants (keywords). Below are the VBScript literal keywords, followed by operators, functions, statements and objects.

VBScript Literals

Literal keywords are used to define variables and constants, or comparison of variables.

VBScript Literal Keywords

Keyword	Description
Empty	Uninitialized variable value, e.g. a variable it is created but no value has been assigned to it, or when a variable value is explicitly set to empty. **Note:** Empty is not the same as Null.
False	Boolean condition that is not correct (false has a value of 0)
IsNothing	Variable is an initialized object.
IsEmpty	Variable is uninitialized.
IsNull	Variable contains invalid data.
Nothing	Indicates an uninitialized object value, or disassociate an object variable from an object to release system resources.
Null	Variable contains no valid data. **Note:** This is not the same as Empty or Nothing
True	Boolean condition that is correct (true has a value of -1)

The following are example uses literal keywords

```
Dim valve_closed, pump_on, a
If valve_closed = True Then pump_on = False
a = Empty
```

VBScript Operators

Operators act on one or more operands. VBScript provides operators to perform arithmetic, assignment, comparison, concatenation and logical operations. In some cases, the operation varies based on the Variant subdata type.

Arithmetic Operators

Arithmetic operator are used to calculate a numeric value, and are normally used with in conjunction with the assignment operator or one of the comparison operators. Note that the minus (**–**) operator can also be a unary operator to indicate a negative number. The plus (**+**) operator can be used for addition of two numbers or to concatenate strings, although the ampersand (**&**) operator is the preferred operator for string concatenation.

Operators	Name	Description
+	Addition	Adds two numbers together
-	Subtraction	Subtracts one number from the other
^	Exponentiation	Raises number to the power of the exponent
Mod	Mod	Divides one number by the other and returns only the remainder
*****	Multiplication	Multiplies two numbers
/	Division	Divides one number by the other with a floating point result
\	Integer Division	Divides one number by the other with an integer result

Assignment Operator

The assignment operator is used to assign a value to a variable or to a property of an object. See the **Set** statement for referencing and assigning Objects.

Operators	Name	Description
=	Assignment	Assign a value to a variable or property, reference and assign objects

Comparison Operators

Comparison operators are used to compare numeric values and string expressions against other variables, expressions or constants. The result of the comparison is either a logical **True** or a logical **False.**

Operators	Name	Example	Description
<	Less than	a < b	Returns TRUE if a < b
<=	Less than or equal	a <= b	Returns TRUE if a is not greater than b
>	Greater than	a > b	Returns TRUE if a is greater than b
>=	Greater than or equal	a >=b	Returns TRUE if a is not less than b
=	Equals	a = b	Returns TRUE if a is equal to b
<>	Not equal	a <> b	Returns TRUE if a is not equal to b

String Concatenation Operators

The String operators are used to concatenate (combine) strings. There are two string concatenation operators, but it is recommended to use the **&** operator for string concatenation, so as not to confuse it with the **+** addition operator.

Operators	Name	Description
&	String Concatenation	Concatenates two strings. Preferred method.
+	String Concatenation	Concatenates two strings. Non-preferred method.

Logical Operators

Logical Operators are used to perform logical operations on expressions, and can also be used as bit-wise operators.

Operators	Function	Example	Returns
And	Logical And	a AND b	True only if a and b are both true
Eqv	Logical Equivalent	a Eqv b	True if a and b are the same
Imp	Logical Implication	a Imp b	False only if a is true and b is false otherwise true
Not	Logical Not	a Not b	True if a is false; False if a is true
Or, |	Logical Or	a OR b	True if a or b is true, or both are true
Xor	Logical Exclusive Or	a Xor b	True if a or b is true, but not both

A word of caution about the NOT operator. The NOT operator inverts boolean "True" and "False" values as expected. However, the NOT operator can also operate on other data subtypes and IWS data types. The chart to the right shows the result of the NOT operation on integer (or Real) values.

Expression	NOT of Expression
True	**False**
False	**True**
3	**-4**
2	**-3**
1	**-2**
0	**-1**
-1	**0**
-2	**1**
-3	**2**

Is Operator

The **Is** operator is used to compare one object variable to another to determine if they reference the same object. In addition, the **Nothing** keyword can be used to determine whether a variable contains a valid object reference..

$ Operator

The **$** operator is a very special operator which has been added by InduSoft. The **$** operator allows VBScript to access IWS tags and built-in functions. IWS tags can be used in expressions similar to VBScript variables. Remember that IWS variable types can be different that VBScript data subtypes.

When the **$** operator is used, Intellisense (part of VBScript) will display all current IWS tags and built-in functions in a scroll-down menu. The developer can choose from any of these, or add new tags by typing a unique name. If a new tag name is entered, IWS will then prompt the developer for tag type specifications.

```
Example:    $Temp1 = 100                 ‘ Sets IWS tag Temp1 to a value 100
            MsgBox $temp1                ‘ Prints the value of IWS tag Temp1
```

Addition Operator (+)

Description	Sums two numbers.
Usage	*result* **=** *expression1* **+** *expression2*
Arguments	*result*

Any numeric or string variable.

expression1

Any valid numeric or string expression.

expression2

Any valid numeric or string expression.

Result — Either numeric or string, depending on the arguments

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*.

See also — **Concatenation Operator (&), Concatenation Operator (+), Subtraction Operator (-)**

Example:

```
a = 5 :
b = 6
c = a + b                       ‘ Variable c is now 11
```

And Operator (And)

Description — Performs a logical conjunction on two expressions to see if both are True

Usage — *result* **=** *expression1* **And** *expression2*

Arguments — *result*

Any variable..

expression1

Any expression.

expression2

Any expression.

Return — If, and only if, both expressions evaluate to **True**, *result* is **True**. If either expression evaluates to **False**, *result* is **False**.

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	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	Then *result* is
0	0	0
0	1	0
1	0	0
1	1	1

Example:

```
Dim pump_on
Dim valve_closed
If ( (pump_on = True) And (valve_closed = True)) then pump_on = False
```

Example

```
a = 5                    ‘ a = 5  (bits 0000 0000 0000 0101)
b = 4                    ‘ b = 4  (bits 0000 0000 0000 0100)
c = a And b              ‘ c = 4  (bitwise AND operation)
```

Assignment Operator (=)

Description Used to assign a value to a variable or a property.

Usage *variable* = *value*

Arguments *variable*

Any variable or 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 writeable at runtime.

See also Comparison Operator, **Set** Statement

Example:

```
a = 5
b = 6
c = a + b              ‘ Variable c is now 11
```

Comparison Operators (<, <=, >, >=, =, <>)

Description Used to compare expressions

Usage *result* **=** expression1 comparisonoperator expression2
Conditional

Arguments *result*
Any numeric variable.
expression
Any expression.
comparisonoperator
Any comparison operator.

Remarks 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	The numeric expression is less than the string other is a string expression.
One expression is **Empty** and the	Perform a numeric comparison, using 0 as the other is numeric **Empty** expression.
One expression is **Empty** and the	Perform a string comparison, using a zero-length
other is a string	string ("") as the **Empty** expression.
Both expressions are **Empty**	The expressions are equal

Example:
```
If a>b then c = c +1
MyResult = a = b                 ‘ If a = b, then MyResult = True, otherwise False
```

Concatenation Operator (&)

Description	Forces string concatenation of two expressions.
Usage	*result* **=** *expression1* **&** *expression2*
Arguments	*result* Any variable.. *expression1* Any expression. *expression2* Any expression.
Return	Result will be converted to a string subtype if it is not already
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. **Note:** In addition to the **&** operator, you can also use the **+** operator for string concatenation, although use of the **&** operator is the preferred method.

Example:

```
Dim str1, str2, str3
str1 = “AB”
str2 = “CD”
str3 = str1 & str2                    ‘ str3 equals “ABCD”
```

Concatenation Operator (+)

Description	Concatenates two strings.
Usage	*result* **=** *expression1* **+** *expression2*
Arguments	*result* Any numeric or string variable. *expression1* Any valid numeric or string expression. *expression2* Any valid numeric or string expression.
Result	Either numeric or string, depending on the arguments
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*.

See also **Addition Operator (+),Concatenation Operator (&), Subtraction Operator (-)**

Example:

```
a = “ABC” :
b = “DEF”
c = a + b                    ‘ Variable c is now “ABCDEF”
a = 1
b = “1”
c = a + b                    ‘ Variable c is now 2 (numeric value)
```

Division Operator (/)

Description	Divides two numbers and returns a floating-point result
Usage	*result* **=** *number1* **/** *number2*
Arguments	*result* Any numeric variable. *number1* Any valid numeric expression. *number2* Any valid numeric expression.
Return	A floating point number.
Remarks	If one or both expressions are Null expressions, *result* is **Null**. Any expression that is Empty is treated as 0.
See also	**Multiplication Operator(*)**, **Integer Division Operator(\\)**
Example:	Dim a, b a = 3 b = A / 2 ‘ The result b is equal to 1.5

Eqv Operator (Eqv)

Description: Performs a logical equivalence on two expressions, checking if both expressions evaluate to the same value

Usage: *result* **=** *expression1* **Eqv** *expression2*

Arguments:

result
Any numeric variable.

expression1
Any expression, must evaluate to True, False, or Null

expression2
Any expression, must evaluate to True, False, or Null

Return: Returns True is both expressions evaluate to the same value (True or False)

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	Then *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	Then *result* is
0	0	1
0	1	0
1	0	0
1	1	1

Example:

Dim cond1, cond2, cond3
cond1 = **False**
cond2 = **False**
cond3 = cond1 **Eqv** cond2 ‘ cond3 is set to True

Example

```
a = 5                    ‘ Bitwise 5= 0000 0000 0000 0101
b = 4                    ‘ Bitwise 4 = 0000 0000 0000 0100
MyResult = a Eqv b       ‘ Result = -2 = 1111 1111 1111 1110
```

Exponentiation Operator (^)

Description	Raises a number to the power of an exponent
Usage	*result* **=** *number* **^** *exponent*
Arguments	*result* Any numeric variable. *number* Any valid numeric expression *exponent* Any valid 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.**

Example:

```
Dim a
a = 2
a = a ^ 2                           ‘ a now equal 4
a = 5 ^ 5                           ‘ a is now 3,125
```

Imp Operator (Imp)

Description	Performs a logical implication on two expressions
Usage	*result* **=** *expression1* **Imp** *expression2*
Arguments	*result* Any 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

Example:

```
Dim cond1, cond2, cond3
cond1 = True
cond2 = Null
cond3 = cond1 Imp cond2             ‘ cond3 is set to Null
```

Example

```
MyResult = 0 Imp 0                  ‘ Returns 1
MyResult = 1 Imp 0                  ‘ Returns 0
MyResult = 1 Imp 1                  ‘ Returns 1
```

Example

```
a = 5                               ‘ Bitwise 0000 0000 0000 0101
b = 4                               ‘ Bitwise 0000 0000 0000 0100
c = a Imp b                         ‘ Bitwise 1111 1111 1111 1110 (-2) result
```

Integer Division Operator (\)

Description	Divides two numbers and returns an integer result
Usage	*result* **=** *number1* \ *number2*
Arguments	*result*
	Any numeric variable.
	number1
	Any valid numeric expression.
	number2
	Any valid numeric expression.
Return	The integer part of the result when dividing two numbers
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.
See also	**Multiplication Operator (*)** and **Division Operator(/)**
Example:	Dim a, b
	a = 3
	b = A \\ 2 ‘ The result b is equal to 1

Is Operator (Is)

Description	Compares two object reference variables
Usage	*result* **=** *object1* **Is** *object2*
Arguments	*result*
	Any numeric variable.
	number1
	Any object name
	number2
	Any object name.
Return	Logical **True** if both objects refer to the same object, otherwise **False**
Remarks	If *object1* and *object2* both refer to the same object, *result* is **True**, otherwise *result* is **False**.
See also	**Set Statement**
Example:	Set obj1 = CreateObject(“ADODB.Connection”)
	Set obje2 = obj1
	MyTest = obj1 **Is** obj2 ‘ Returns a true
	MyTest = obj1 **Not Is** obj2 ‘ Returns false
	MyTest = obj1 **Is Nothing** ‘ Checks to see if object is valid. Returns False

Modulus Division Operator (Mod)

Description	Divides two numbers and returns only the remainder.
Usage	*result* **=** *number1* **Mod** *number2*
Arguments	*result*
	Any numeric variable.
	number1
	Any valid numeric expression.
	number2
	Any valid 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*. **Bytes**, **Integers** and **Long** subtype are rounded to the smallest possible subtype before the Modulus division is performed. If any expression is **Null**, *result* is also **Null**. Any expression that is Empty is treated as 0.
Example:	A = 19 Mod 6.7 ‘ The result A equals 5

Multiplication Operator (*)

Description	Multiplies two numbers
Usage	*result* **=** *number1* * *number2*
Arguments	*result* Any numeric variable. *number1* Any valid numeric expression. *number2* Any valid 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.
See also	**Division Operator (/)**, **Integer Division (\)**

Example:	Dim A, B, C
	A = 2 : B = 3
	C = A * B ‘ The result C is equal to 6

Not Operator (Not)

Description Performs a logical Not on an expression

Usage *result* **= Not** *expression*

Arguments *result*

Any variable.

expression

Any valid expression

Returns A logical **True** or **False**

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	0
1	1

See also **AND Operator**, **OR Operator**, **XOR Operator**

Example:

```
Dim cond1, cond2, a
cond1 = True
cond2 = Not cond1                ‘ cond2 set to False
```

Example

```
a = 5                            ‘ a = 5    (bit   0000 0000 0000 0101)
a = Not a                        ‘ a = -6   (bit   1111 1111 1111 1010)
```

Or Operator (Or, |)

Description — Performs a logical disjunction on two expressions.

Usage — *result* **=** *expression1* **Or** *expression2*

Arguments — *result*
    Any variable.
expression1
    Any valid expression.
expression2
    Any valid 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

See also — **AND Operator**, **NOR Operator**, **XOR Operator**

Example:

```
Dim alarm1, alarm2
Dim alarm_light
If ( (alarm1 = True) Or (alarm2 = True)) then
    alarm_light = True
    Else
    alarm_light = False
End If
```

Example

```
a = 5                          ‘ a = 5 (bitwise 0000 0000 0000 0101)
b = 4                          ‘ b = 4 (bitwise 0000 0000 0000 0100)
MyResult = a Or b              ‘ Result = 5
```

Subtraction Operator (-)

Description	Finds the difference between two numbers or indicates the negative value of a numeric expression.

Usage

result **=** *number1* **-** *number2* (Syntax 1)
result = - *number* (Syntax 2)

Arguments

result
Any numeric variable.
number1
Any valid numeric expression.
number2
Any valid numeric expression.
number
Any valid numeric value or 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.

Example:

```
MyResult = 5 - 4                    ' MyResult is 1
a = 5
MyResult = -a                       ‘ MyResult is -5
MyResult = -(5-4)                   ‘ MyResult is -1
```

Xor Operator (Xor)

Description

Performs a logical exclusion on two expressions.

Usage

result **=** *expression1* **Xor** *expression2*

Arguments

result
Any variable.
expression1
Any valid expression.
expression2
Any valid 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

See also **And Operator**, **Not Operator**, **Or Operator**

Example:

```
Dim cond1, cond2, flag
cond1 = True
cond2 = False
If (cond1 = True) Xor (cond2 = True) Then
    flag = True
    Else
    flag = False
End If
```

Example

```
a = 5                       ‘ Bitwise 0000 0000 0000 0101
b = 4                       ‘ Bitwise 0000 0000 0000 0100
c = a Xor b                 ‘ Bitwise 0000 0000 0000 0001 result (=1)
```

Operator Precedence

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	&

Highest Priority $\longrightarrow$ Lowest Priority

Highest Priority $\downarrow$ Lowest Priority

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 its precedence does fall in 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 only checks to determine if two object references refer to the same object.

VBScript Functions

Function Summary

Functions			
Abs	**Escape**	**LCase**	**Second**
Array	**Eval**	**Left**	**SetLocale**
Asc	**Exp**	**LeftB**	**Sgn**
AscB	**Filter**	**Len**	**Sin**
AscW	**Fix**	**LenB**	**Space**
Atn	**FormatCurrency**	**LoadPicture**	**Split**
CBool	**FormatDateTime**	**Log**	**Sqr**
CByte	**FormatNumber**	**LTrim**	**StrComp**
CCur	**FormatPercent**	**Mid**	**String**
CDate	**GetLocale**	**MidB**	**StrReverse**
CDbl	**GetObject**	**Minute**	**Tan**
Chr	**GetRef**	**Month**	**Time**
ChrB	**Hex**	**MonthName**	**Timer**
ChrW	**Hour**	**MsgBox**	**TimeSerial**
CInt	**InputBox**	**Now**	**TimeValue**
CLng	**InStr**	**Oct**	**Trim**
Cos	**InstrB**	**Replace**	**TypeName**
CreateObject	**InStrRev**	**RGB**	**UBound**
CSng	**Int**	**Right**	**UCase**
CStr	**IsArray**	**RightB**	**Unescape**
Date	**IsDate**	**Rnd**	**VarType**
DateAdd	**IsEmpty**	**Round**	**Weekday**
DateDiff	**InNull**	**RTrim**	**WeekdayName**
DatePart	**IsNumeric**	**ScriptEngine**	**Year**
DateSerial	**IsObject**	**ScriptEngineBuildVersion**	
DateValue	**Join**	**ScriptEngineMajorVersion**	
Day	**LBound**	**ScriptEngineMinorVersion**	

VBScript Array Functions

Function	Description
Array	Returns a variant containing an array
Filter	Returns a zero-based array that contains a subset of a string array based on a filter criteria
IsArray	Returns a Boolean value that indicates whether a specified variable is an array
Join	Returns a string that consists of a number of substrings in an array
LBound	Returns the smallest subscript for the indicated dimension of an array
Split	Returns a zero-based, one-dimensional array that contains a specified number of substrings
UBound	Returns the largest subscript for the indicated dimension of an array

Note: See VBScript Statements as well

VBScript Object Functions

Function	Description
CreateObject	Creates and returns a reference to an Automation object
GetObject	Returns a reference to an Automation object from a file
IsObject	Returns a Boolean value indicating whether an expression references a valid Automation object.

Note: See VBScript Objects and Collections as well

VBScript Math Functions

Function	Description
Abs	Returns the absolute value of a specified number
Atn	Returns the arctangent of a specified number
Cos	Returns the cosine of a specified number (angle)
Exp	Returns *e* raised to a power
Hex	Returns the hexadecimal value of a specified number
Int	Returns the integer part of a specified number
Fix	Returns the integer part of a specified number
Log	Returns the natural logarithm of a specified number
Oct	Returns the octal value of a specified number
Randomize	Initializes the random-number generator
Rnd	Returns a random number less than 1 but greater or equal to 0
Sgn	Returns an integer that indicates the sign of a specified number
Sin	Returns the sine of a specified number (angle)
Sqr	Returns the square root of a specified number
Tan	Returns the tangent of a specified number (angle)

Note: See VBScript Derived Functions as well

VBScript String Functions

Function	Description
Escape	Encodes a string so it contains only ASCII characters
InStr	Returns the position of the first occurrence of one string within another. The search begins at the first character of the string
InStrB	Returns the position of the first occurrence of one string within another. The search begins at the first byte of the string
InStrRev	Returns the position of the first occurrence of one string within another. The search begins at the last character of the string
LCase	Converts a specified string to lowercase
Left	Returns a specified number of characters from the left side of a string
LeftB	Returns a specified number of bytes from the left side of a string
Len	Returns the number of characters in a string
LenB	Returns the number of bytes in a string
LTrim	Removes spaces on the left side of a string
Mid	Returns a specified number of characters from a string
MidB	Returns a specified number of bytes from a string
Replace	Replaces a specified part of a string with another string a specified number of times
Right	Returns a specified number of characters from the right side of a string
RightB	Returns a specified number of bytes from the right side of a string
RTrim	Removes spaces on the right side of a string
Space	Returns a string that consists of a specified number of spaces
StrComp	Compares two strings and returns a value that represents the result of the comparison
String	Returns a string that contains a repeating character of a specified length
StrReverse	Reverses a string
Trim	Removes spaces on both the left and the right side of a string
UCase	Converts a specified string to uppercase
UnEscape	Decodes a string encoded with the Escape function

VBScript Conversions Functions

Function	Description
Abs	Returns the absolute value of a specified number
Asc	Converts the first letter in a string to ANSI code
CBool	Converts an expression to a variant of subtype Boolean
CByte	Converts an expression to a variant of subtype Byte
CCur	Converts an expression to a variant of subtype Currency
CDate	Converts a valid date and time expression to the variant of subtype Date
CDbl	Converts an expression to a variant of subtype Double
Chr	Converts the specified ANSI code to a character
CInt	Converts an expression to a variant of subtype Integer
CLng	Converts an expression to a variant of subtype Long
CSng	Converts an expression to a variant of subtype Single
CStr	Converts an expression to a variant of subtype String
Fix	Returns the integer part of a specified number
Hex	Returns the hexadecimal value of a specified number
Int	Returns the integer part of a specified number
Oct	Returns the octal value of a specified number
Round	Returns a rounded number
Sgn	Returns the integer portion of a number

VBScript Format Functions

Function	Description
FormatCurrency	Returns an expression formatted as a currency value
FormatDateTime	Returns an expression formatted as a date or time
FormatNumber	Returns an expression formatted as a number
FormatPercent	Returns an expression formatted as a percentage

VBScript Time and Date Functions

Function	Description
CDate	Converts a valid date and time expression to the variant of subtype Date
Date	Returns the current system date
DateAdd	Returns a date to which a specified time interval has been added
DateDiff	Returns the number of intervals between two dates
DatePart	Returns the specified part of a given date
DateSerial	Returns the date for a specified year, month, and day
DateValue	Returns a date
Day	Returns a number that represents the day of the month (between 1 and 31, inclusive)
FormatDateTime	Returns an expression formatted as a date or time
Hour	Returns a number that represents the hour of the day (between 0 and 23, inclusive)
IsDate	Returns a Boolean value that indicates if the evaluated expression can be converted to a date
Minute	Returns a number that represents the minute of the hour (between 0 and 59, inclusive)
Month	Returns a number that represents the month of the year (between 1 and 12, inclusive)
MonthName	Returns the name of a specified month
Now	Returns the current system date and time
Second	Returns a number that represents the second of the minute (between 0 and 59, inclusive)
Time	Returns the current system time
Timer	Returns the number of seconds since 12:00 AM
TimeSerial	Returns the time for a specific hour, minute, and second
TimeValue	Returns a time
Weekday	Returns a number that represents the day of the week (between 1 and 7, inclusive)
WeekdayName	Returns the weekday name of a specified day of the week
Year	Returns a number that represents the year

VBScript Expression Functions

Expressions	Description
Eval	Evaluates an expression and returns the result

Note: See VBScript Objects and Collections as well

VBScript I/O Functions

Input/Output	Description
InputBox	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.
MsgBox	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.
LoadPicture	Returns a picture object

VBScript Script Engine Functions

Script Engine ID	Description
ScriptEngine	Returns a string representing the scripting language in use
ScriptEngineBuildVersion	Returns the build version number of the scripting engine in use
ScriptEngineMajorVersion	Returns the major version number of the scripting engine in use
ScriptEngineMinorVersion	Returns the minor version number of the scripting engine in use

VBScript Variant Functions

Variants	Description
IsArray	Returns a Boolean value indicating whether a variable is an array
IsDate	Returns a Boolean value indicating whether an expression can be converted to a date
IsEmpty	Returns a Boolean value indicating whether a variable has been initialized.
IsNull	Returns a Boolean value that indicates whether an expression contains no valid data (Null).
IsNumeric	Returns a Boolean value indicating whether an expression can be evaluated as a number
IsObject	Returns a Boolean value indicating whether an expression references a valid Automation object.
TypeName	Returns a string that provides Variant subtype information about a variable
VarType	Returns a value indicating the subtype of a variable

VBScript Miscellaneous Functions

Miscellaneous	Description
RGB	Returns a whole number representing an RGB color value
GetLocale	Returns the current locale ID
SetLocale	Sets the current locale ID

VBScript Functions

Abs

Description	Returns the absolute value of a number
Usage	*result* = **Abs(***number***)**
Arguments	*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.
Returns	The absolute value of a number is its unsigned magnitude. The data type returned is the same as that of the *number* argument.
Remarks	**Abs(**-1**)** and **Abs(**1**)** both return 1.
Example	myNumber = **Abs**(-50.3) ‘ Returns 50.3

Array

Description	Returns a Variant containing an subtype array
Usage	*varArray* = **Array** (*arglist*)
Arguments	*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. All arrays are zero-based, meaning that the first element in the list will be element 0 in the Array.
Returns	Returns a **Variant** array
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. 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.
See also	Dim, Erase
Example	**Dim** A
	A = **Array** (10, 20, 30)
	B = A(2) ‘ B is now 30

Asc

Description	Finds the ANSI character code corresponding to the first letter in a string
Usage	intCode = **Asc(***string***)**
Arguments	*string*
	The *string* argument is any valid string expression.
Returns	Returns an integer code representing the ANSI character code corresponding to the first letter in a string
Remarks	If the string expression contains no characters, a run-time error occurs. *string* is converted to a String subtype if it contains numeric data.
See also	**AscB, AscW, Chr, ChrB, ChrW**
Example	Dim myNumber
	myNumber = Asc(“A”) ‘ Returns 65
	myNumber = Asc(“a”) ‘ Returns 97
	mynumber = Asc(“Apple”) ‘ Returns 65

AscB

Description	Returns the ANSI character code for the first byte in a string of byte data
Usage	intCode = **AscB**(*string*)
Arguments	*string* The string argument is any valid string expression.
Return	Returns an integer code representing the ANSI character code corresponding to the first byte in a string containing byte data
Remarks	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. Remember that Intel machines use little endian (i.e. in a double word, the least significant byte is first, then the most significant). Motorola architectures are big endian (most significant byte first). If the string contains no characters, a run-time error occurs. For normal ANSI (8-bit) strings, this function will return the same result as the **Asc** function. Only if the string is in UniCode (16-bit) format will it be different from the **Asc** function. A runtime error will occur if *string* does not contain any characters.
See also	**Asc, AscW, Chr, ChrB, ChrW**

Example

```
inBuffer = “”                              ‘Null out the buffer string
For k = 1 To 4                             ‘Load a buffer string
    inBuffer = inBuffer & k                ‘will have the characters 1, 2, 3, 4
Next
myStr = “”                                 ‘null out buffer
For k = 1 To LenB(inBuffer) Step 2         ‘Get buffer length, every char = 2 bytes
    myStr = myStr & Hex(AscB(MidB(inBuffer, k, 1)))
Next                                       ‘get the individual character, convert it to an ASCII
                                           ‘value, then show the hex equivalent
MsgBox myStr                               ‘Displays 31323334
```

AscW

Description	Returns the UniCode character code for the first character in a string
Usage	intCode = **AscW**(*string*)
Arguments	*String* The string argument is any valid string expression. If the string contains no characters, a run-time error occurs
Return	Returns an integer code representing the UniCode character code corresponding to the first letter in a string.
Remarks	**AscW** is provided for 32-bit platforms that use Unicode characters. It returns a Unicode (16-bit) character code, thereby avoiding the conversion from Unicode to ANSI. A runtime error will occur if *string* does not contain any characters. *string* is converted to a String subtype if it contains numeric data.
See also	**Asc, AscB, Chr, ChrB, ChrW**

Example

```
in_buffer = “Ö”                            ‘Unicode character Ö in buffer
MsgBox AscW(in_buffer)                     ‘Displays 214 (decimal)
MsgBox Hex(AscW(in_buffer))                ‘Displays D6 (hexadecimal)
```

Atn

Description	Returns the arctangent of a number
Usage	realRslt = **Atn**(*number*)
Arguments	*number* The number argument can be any valid numeric expression.
Return	Returns the arctangent of a *number* as Variant subtype Double. Result is in radians.
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. **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. **Atn** is not to be confused with the cotangent, which is the simple inverse of a tangent (1/tangent).
See also	**Cos, Sin, and Tan**
Example	**Dim** pi pi = 4 * **Atn**(1) ' Calculate the value of pi.

CBool

Description	Returns an expression that has been converted to a Variant of subtype Boolean
Usage	boolRslt = **CBool**(*expression*)
Arguments	*expression* Any valid expression
Return	Boolean value corresponding to the value of the expression
Remarks	If the expression is zero, **False** is returned; otherwise **True** is returned. If the *expression* cannot be interpreted as a numeric value, a run-time error occurs.
See also	**CByte, CCur, CDbl, CInt, CLng, CSng, CStr**

Example	Dim A, B, Check
	A = 5
	B= 5
	Check = CBool (A = B) ‘ Check contains **True**
	A= 0
	Check = CBool (A) ‘ Check contains **False**

CByte

Description	Returns an expression that has been converted to a Variant of subtype Byte
Usage	byteVal = **CByte** (*expression*)
Arguments	*expression* The *expression* argument is any valid numeric expression
Return	An expression converted to a Byte value
Remarks	A runtime error occurs if *expression* can’t be evaluated to a numeric value. If *expression* lies outside the acceptable range for the byte subtype (0-255), an Overflow error occurs. If expression is a floating point number, it is rounded to the nearest integer and then converted to byte subtype. 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.
See also	**CBool, CCur, CDbl, CInt, CLng, CSng, CStr**

Example	Dim myDouble, myByte
	myDouble = 123.45678
	myByte = CByte(myDouble) ‘ myByte contains 123

CCur

Description	Returns an expression that has been converted to a **Variant** of subtype **Currency**
Usage	curVal = **CCur**(*expression*)
Arguments	*expression* The *expression* argument is any valid expression
Return	An expression converted to a Currency value
Remarks	**CCur** provides an internationally aware conversion from any data type to a **Currency** subtype. The return value is based on the locale settings on the local PC. For example, different decimal separators and thousands separators are properly recognized depending on the locale setting of your system.
See also	**CBool, CByte, CDbl, CInt, CLng, CSng, CStr**

Example	Dim myDouble, myCurr
	myDouble = 543.214588 'myDouble is a Double.
	myCurr = **CCur(**myDouble * 2**)** 'Multiply by * 2 and convert
	MsgBox myCurr ‘Result 1086.4292 (based on local PC settings)

```
Dim myDouble, myCurr
myDouble = 543.214588                    'myDouble is a Double.
myCurr = CCur(myDouble * 2)              'Multiply by * 2 and convert
MsgBox myCurr                            ‘Result 1086.4292 (based on local PC settings)
```

CDate

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

Usage
dateVal = **CDate**(*date*)

Arguments
The date argument is any valid date expression, of numeric or string type .

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.

If your PC is set to the American date format (mm/dd/yy), and you enter the British date format (dd/mm/yy) in a text box, the **CDate** function will convert it to the American mm/dd/yy format.
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.

See also
IsDate, DateValue, TimeValue

Example
```
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
myShortDate = CDate(#04/18/2006#)        ‘myShortDate holds value 4/18/2006
```

CDbl

Description	Returns an expression that has been converted to a **Variant** of subtype **Double**.
Usage	dblVal = **CDbl(***expression***)**
Arguments	*expression* The *expression* argument is any valid expression
Return	An expression converted to a double precision real value
Remarks	**CDbl** provides an internationally aware conversion from any data type to a **Double (double precision real)** subtype. The return value is based on the locale settings on the local PC. For example, different decimal separators and thousands separators are properly recognized depending on the locale setting of your system.
See Also	**CBool, CByte, CCur, CInt, CLng, CSng, CStr**
Example	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).

Chr

Description	Returns the ANSI character corresponding to a character code
Usage	strChar = **Chr**(*charcode*)
Arguments	*charcode* The *charcode* argument is a numeric value that identifies the character
Return	An ANSI character (string)
Remarks	Numeric values 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:
See Also	**Asc, AscB, AscW, ChrB, ChrW**

Example	Dim myChar
	myChar = Chr(65) 'Returns A
	myChar = Chr(97) 'Returns a
	mychar = Chr(37) 'Returns %

ChrB

Description	Returns the ANSI character corresponding to a character code contained in a byte data string.
Usage	strChar = **ChrB**(*bytecode*)
Arguments	*bytecode* The *bytecode* argument is a numeric value that indicates the character
Return	This function is used instead of the **Chr** function when you want only the first byte of the character returned. Numeric values from 0 to 31 are the same as standard, nonprintable ASCII codes. For example, **ChrB(**10**)** returns a linefeed character.
Remarks	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.
See Also	**Asc, AscB, AscW, Chr, ChrW**
Example	Dim myChar myChar = Chr(89) 'Returns Y

ChrW

Description	Returns the UniCode character corresponding to a character code
Usage	strChar = **ChrW**(*charcode*)
Arguments	*charcode* The *charcode* argument is a numeric value that indicates the character
Return	A UniCode character
Remarks	**ChrW** is used instead of the **Chr** or **ChrB** functions to return a 2-byte UniCode character. **ChrW** is provided for 32-bit platforms that use Unicode characters.
See Also	**Asc, AscB, AscW, Chr, ChrB**
Example	Dim myChar myChar = ChrW(214) ‘Returns Ö

CInt

Description	Returns an expression that has been converted to a **Variant** of subtype **Integer.**
Usage	intVal = **CInt(***expression*)
Arguments	*expression* The *expression* argument is any valid expression
Return	An integer value
Remarks	**CInt** provides an internationally aware conversion 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. **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.
See Also	**CBool, CByte, CCur, CDbl, CLng, CSng, CStr**

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

CLng

Description	Returns an expression that has been converted to a **Variant** of subtype **Long.**
Usage	LngVal = **CLng(***expression*)
Arguments	*expression* The *expression* argument is any valid expression
Return	A long integer value
Remarks	**CLng** provides an internationally aware conversion 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. **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.
See Also	**CBool, CByte, CCur, CDbl, CInt, CSng, CStr**

Example	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.

Cos

Description	Returns the cosine of an angle.
Usage	realVal = **Cos**(*number*)
Arguments	*number* The number argument can be any valid numeric expression that expresses an angle in radian
Return	Returns the cosine of an angle as a Variant subtype Double. Result is 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.
See also	**Atn, Sin,** and **Tan**
Example	see below

```
Dim MyAngle, MySecant
MyAngle = 1.3                    ' Define angle in radians.
MySecant = 1 / Cos(MyAngle)      ' Calculate secant.
Angle = (30 * 3.14159/180)       ‘ Convert 30 degrees into radians
AngleCos = Cos(Angle)            ‘ Compute cosine of angle
```

CreateObject

Description: Creates and returns a reference to an Automation object.

Usage: Set objName = **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.

Return: An object reference

Remarks: The *servername* and *typename* together are often referred to as a ProgID, or Programmatic ID. A ProgID may actually have multiple parts (e.g. *servername.typename.version*)To avoid confusion, note that the parameter *servername* refers to a Microsoft COM server (automation server) applications such as Microsoft Access, Excel, Word. Other COM servers such as ADO.NET can be referenced. 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. This code starts the application that creates the object (in this case, a Microsoft Excel spreadsheet).

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

Once an object is created, refer to it in VBScript 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.

```
ExcelSheet.Application.Visible = True                ' Make Excel object. visible
ExcelSheet.ActiveSheet.Cells(1,1).Value = "ABC"      ‘ Place text in row 1, col 1
ExcelSheet.SaveAs "C:\DOCS\TEST.XLS"                 ' Save the sheet.
ExcelSheet.Application.Quit                          ‘ Close Excel
Set ExcelSheet = Nothing                             ' Release the object variable.
```

Creating an object on a remote server can only be accomplished when Internet security is turned off. You can create an object on a remote networked computer by passing the name of the computer to the *servername* argument of **CreateObject**. That name is the same as the machine name portion of a share name. For a network share named "\\myserver\public", the *servername* is "myserver". In addition, you can specify *servername* using DNS format or an IP address.

Example: The following code returns the version number of an instance of Excel running on a remote network computer named "myserver". An error occurs if the specified remote server does not exist or cannot be found.

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

CSng

Description	Returns an expression that has been converted to a **Variant** of subtype **Single**
Usage	sngVal = **CSng(***expression*)
Arguments	*expression* The *expression* argument is any valid expression
Return	A single-precision real value
Remarks	**CSng** provides an internationally aware conversion 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
See Also	**CBool, CByte, CCur, CDbl, CInt, CLng, CStr**

Example

```
Dim MyDouble1, MyDouble2,
Dim 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.
```

CStr

Description	Returns an expression that has been converted to a **Variant** of subtype **String**
Usage	strVal = **CStr(***expression*)
Arguments	*expression* The *expression* argument is any valid expression
Return	A string value
Remarks	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

See Also **CBool, CByte, CCur, CDbl, CInt, CLng, CSng**

Example

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

Date

Description	Returns a **Variant** of subtype **Date** indicating the current system date.
Usage	dateVal = **Date**
Arguments	none
Return	Returns a **Variant** subtype **Date**
Remarks	The locale setting can be specified to use the dash “-“ or the forward slash “/” as a separator
See Also	**Now, Time**

Example

```
Dim Mydate
MyDate = Date          ‘ Mydate contains the current system date
MsgBox Date
```

DateAdd

Description	Returns a date to which a specified time interval has been added or subtracted
Usage	dateVal = **DateAdd(***interval***,** *number***,** *date***)**
Arguments	*interval* Required. String expression that is the interval you want to add. . *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 **Date** 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

Return — A Date value

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. 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.
DateAdd is internationally aware, meaning the return value is based on the locale setting on the local machine. Included in the locale settings are the appropriate date and time separators, the dates in the correct order of day, month and year.

See Also — **DateDiff, DatePart**

Example — The following example adds one month to January 31: 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.

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

DateDiff

Description	Returns the number of intervals between two dates
Usage	intVal = **DateDiff**(*interval*, *date1*, *date2* [,*firstdayofweek*[, *firstweekofyear*]])

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 for different language and locale specific settings
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 for different language and locale specific settings
vbFirstJan1	1	Start with the week in which Jan 1 occurs (default)
vbFirstFourDays	2	Start with the week that has at least 4 days in the new year
vbFirstFullWeek	3	Start with the first fill 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.

See Also **DateAdd, DatePart**

Example 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
```

DatePart

Description Returns the specified part of a given date.

Usage **DatePart**(*interval***,** *date*[**,** *firstdayofweek*[**,** *firstweekofyear*]])

Arguments *interval*

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

date

Required. Date expression you want to evaluate.

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 Jan 1 occurs (default)
vbFirstFourDays	2	Start with the week that has at least 4 days in the new year
vbFirstFullWeek	3	Start with the first fill 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.

See Also **DateAdd, DateDiff**

Example 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
```

DateSerial

Description	Returns a **Variant** of subtype **Date** for a specified year, month, and day
Usage	dateVal = **DateSerial**(*year, month, day*)
Arguments	*year* Any numeric value or expression that evaluates to a number between 100 and 9999 *month* Any numeric value or expression that evaluates to a number between 1 and 12 *day* Any numeric value or expression that evaluates to a number between 1 and 31
Return	A date value
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. 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. The same is true for negative values and the value 0, but instead of incrementing, the next larger unit is decremented. 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.
See Also	**Date, DateValue, Day, Month, Now, TimeSerial, TimeValue, Weekday, Year**
Example	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.
```

DateValue

Description	Returns a **Variant** of subtype **Date**
Usage	dateVal = DateValue(*date*)
Arguments	*date* Date is an expression representing a date, time or both, in the range January 1, 100 to December 31, 9999.
Return	A date value
Remarks	Time information in date is not returned. However, if date includes invalid time information (such as "89:98"), a runtime error occurs. **DateValue** is internationally aware and uses the system locale setting on the local machine to recognize the order of a date with only numbers and a separator. If date is a string that includes only numbers separated by valid date separators, **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.
See Also	**Date, DateSerial, Day, Month, Now, TimeSerial, TimeValue, Weekday, Year**
Example	The following example uses the **DateValue** function to convert a string to a date. You can also use date a date to a **Variant** variable, for example, MyDate = #9/11/63#.

```
Dim MyDate
MyDate = DateValue("September 11, 1963")      ' Return a date 9/11/1963
```

Day

Description	Returns a whole number between 1 and 31, inclusive, representing the day of the month
Usage	intVal = **Day**(*date*)
Arguments	*date* The *date* argument is any valid date expression.
Return	An integer value representing the day of the month (1-31).
Remarks	A runtime error occurs if *date* is not a valid expression. If *date* contains Null, **Null** is returned
See Also	**Date, DateSerial, DateValue, Month, Now, TimeSerial, TimeValue, Weekday, Year**

Example

```
Dim MyDay
MyDay = Day (“October 19, 1962”)                    ‘ MyDay contains 19
```

Escape

Description	Encodes a string so it contains only ASCII characters
Usage	strChar = **Escape**(*charString*)
Arguments	*charString* Required. String expression to be encoded.
Remarks	The **Escape** function returns a string (in ASCII format) that contains the contents of charString. All spaces, punctuation, accented characters, and other non-ASCII characters are replaced with %xx encoding, where xx is equivalent to the hexadecimal number representing the character. Unicode characters that have a value greater than 255 are stored using the %uxxxx format
See Also	**UnEscape**

Example

```
Dim cs
cs = Escape(“aÖ”)                                   ‘Returns “a%D6”
```

Eval

Description	Evaluates an expression and returns the result
Usage	boolVal = **Eval(**expression**)**
Arguments	*expression* Required. String containing any legal VBScript expression
Returns	A boolean value
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, the result is **True**; if they are not, the result is **False**. The **Eval** method always uses the second interpretation, whereas the **Execute** statement always uses the first
See Also	**Execute**

Example

```
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
```

Exp

Description	Returns *e* (the base of natural logarithms) raised to a power
Usage	realVal = **Exp**(*number*)
Arguments	*number* The *number* argument can be any valid numeric expression
Return	Returns a **Variant** subtype **Double**
Remarks	If the value of number exceeds 709.782712893, a runtime error occurs. The constant *e* is approximately 2.718282. The **Exp** function complements the action of the **Log** function and is sometimes referred to as the antilogarithm.
See also	**Log**

Example

```
Dim MyAngle, MyHSin
MyAngle = 1.3                                      'Define angle in radians.
MyHSin = (Exp(MyAngle) - Exp(-1 * MyAngle)) / 2    'Calculate hyperbolic sine. Result 1.69838
```

Filter

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

Usage: strArray = **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. Default is **True**

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

Return: A string array

Remarks: If no matches of Value are found within InputStrings, **Filter** returns an empty array. An error occurs if InputString 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. You can use the **Ubound** function to determine the size of the zero-based array returned.

Example: 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".
MyIndex = Filter(MyArray, “n”)             ‘MyIndex(0) contains “Sunday”
                                           ‘MyIndex(1) contains “Monday”
MyIndex = Filter(MyArray, “n”, False)      ‘MyIndex(0) contains “Tuesday”
```

Fix

Description	Returns the integer portion of a number
Usage	intVal = **Fix(***number***)**
Arguments	*number* The number argument can be any valid numeric expression.
Return	An integer value
Remarks	If *number* contains Null, **Null** is returned. **Fix** is internationally aware, meaning the return value is based on the Locale setting on the PC. The data type will be determined from the size of the Integer part. Possible return data types are Integer, Long, Double. 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*)**).**
See also	**Int, Round, CInt, CLng**
Example	see below

```
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.
```

FormatCurrency

Description	Formats an expression as a currency value using the currency symbol defined in the system control panel.
Usage	curValue = **FormatCurrency**(*Expression*[**,***NumDigitsAfterDecimal* [**,***IncludeLeadingDigit* [**,***UseParensForNegativeNumbers* [**,***GroupDigits*]]]])

Arguments

Expression
Required. Any valid 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. Can use one of the following constants:
vbUseDefault 2 Uses settings from the Number tab in Control Panel
vbTrue -1
vbFalse 0

UseParensForNegativeNumbers
Optional. Tristate constant that indicates whether or not to place negative values within parentheses. Can use one of the following constants:
vbUseDefault 2 Uses settings from the Number tab in Control Panel
vbTrue -1
vbFalse 0

GroupDigits
Optional. Tristate constant that indicates whether or not numbers are grouped using the group delimiter specified in the computer's regional settings. Can use one of the following constants:
vbUseDefault 2 Uses settings from the Number tab in Control Panel
vbTrue -1
vbFalse 0

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

Return

Returns **Currency** value

Remarks

When one or more optional arguments are omitted, values for omitted arguments are provided by the computer's regional settings. The position of the currency symbol relative to the currency value is determined by the system's regional settings.

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

See also

FormatDateTime, FormatNumber, FormatPercent

Example

```
Dim MyCurrency
MsgBox  FormatCurrency(1000,2)            ' Displays $1,000.00
MyCurrency = FormatCurrency(1000,2)       ‘ MyCurrency contains “$1,000.00”
```

FormatDateTime

Description	Returns an string formatted as a date or time
Usage	dateVal = **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).

Return	A string formatted as a date and/or time.
Remarks	A runtime error occurs if date is not a valid expression. **Null** will be returned if date contains **Null.** **FormatDateTime** will use the locale settings to determine the format of the date display.
See Also	**FormatCurrency, FormatNumber, FormatPercent**
Example	see below

Example

```
Function GetCurrentDate
    Dim GetCurrentDate
    GetCurrentDate = FormatDateTime(Date, 1)    ' Formats Date into long date.
    Msgbox FormatDateTime(Now, vbShortDate)
End Function
```

FormatNumber

Description	Returns an expression formatted as a number.
Usage	realVal = **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.

Return
A real number either **Single** or **Double** subVariant type

Remarks
When one or more of the optional arguments are omitted, the values for omitted arguments are provided by the computer's regional settings. All settings information comes from the Regional Settings Number tab (locale setting).

See Also
FormatCurrency, FormatDateTime, FormatPercent

Example
```
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
```

FormatPercent

Description	Returns an expression formatted as a percentage (multiplied by 100) with a trailing % character
Usage	realvar = **FormatPercent**(*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.

Return	A real number either **Single** or **Double** subVariant type
Remarks	When one or more of the optional arguments are omitted, the values for omitted arguments are provided by the computer's regional settings. All settings information comes from the Regional Settings Number tab.
See Also	**FormatCurrency, FormatDateTime, FormatNumber**
Example	Dim MyPercent MyPercent = FormatPercent(2/32) ' MyPercent contains 6.25%.

GetLocale

Description	Returns the current locale ID value
Usage	intVal = **GetLocale**()
Arguments	None.
Return	An integer value that determines locale
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. Refer to the Locale ID chart.
See Also	**SetLocale**
Example	MyLocale = GetLocale '.

GetObject

Description	Returns a reference to an Automation object from a file.

Usage	objName = **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.objecttype* and has these parts:

appname

Required; String. Name of the application providing the object.

objecttype

Required; String. Type or class of object to create.

Remarks

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.

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.

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"
```

See Also	**CreateObject**

Example	See Remarks

GetRef

Description	Returns a reference to a DHTML procedure that can be bound to an event
Usage	**Set** object.*eventname* = **GetRef**(*procname*)
Arguments	*object* The name of a DHTML object to which a DHTML 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.
Return	A reference to a DHTML procedure
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. **Note: This function has limited applicability when used with IWS.**

Example:

```
Function GetRefTest()
    Dim Splash
    Splash = "GetRefTest Version 1.0"   & vbCrLf
    Splash = Splash & Chr(169) & " YourCompany"
End Function
Set Window.Onload = GetRef("GetRefTest")
```

Hex

Description	Returns a string representing the hexadecimal value of a number.
Usage	strVal = **Hex**(*number*)
Arguments	*number* The *number* argument is any valid expression.
Return	A **String Variant.**
Remarks	Returns up to 8 characters. If *number* is not already a whole number, it is rounded to the nearest whole number before being evaluated. **Null** will be returned if *number* is Null.

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

See Also: Oct

Example

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

Hour

Description	Returns a whole number between 0 and 23, inclusive, representing the hour of the day.
Usage	intVal = **Hour**(*time*)
Arguments	*time* The time argument is any expression that can represent a time.
Return	An integer value between 0 and 23
Remarks	A runtime error occurs if *time* is not a valid time expression. If *time* contains Null, **Null** is returned.
See Also	**Date, Day, Minute, Month, Now, Second, Weekday, Year**
Example	Dim MyTime, MyHour MyTime = Now MyHour = Hour(MyTime) ‘Contains the number representing the current hour.

InputBox

Description	Displays a dialog box with a custom prompt, waits for the user to input text or click a button, and returns the contents of the text box.
Usage	strRet = **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 ("").
See Also	**MsgBox**
Example	Dim myInput myInput = InputBox("Enter your name") MsgBox ("You entered: " & myInput)

InStr

Description	Returns an integer indicating the position of the first occurrence of one string within another.
Usage	intVal = **InStr**([*start*,]*string1*, *string2*[, *compare*])

Arguments

start
: Optional. Is any valid non-negative numeric expression that indicates the starting position for each search. Non-integer values are rounded. If omitted, search begins at the first character position. 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. (default)
vbTextCompare	1	Perform a textual comparison

Return

An integer value indicating the character position.

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 **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. If *start* contains Null, a runtime error occurs. If *start* is larger than the length of *string2* (*start*>**Len**(*string2*)), 0 will be returned.

See Also

InStrB, InStrRev

Example

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)  ‘Comparison starting at position 4. Returns 6.
MyPos = Instr(1, SearchString, SearchChar, 0)  ‘Comparison starting at position 1. Returns 9.
MyPos = Instr(,SearchString, SearchChar)       ‘Comparison is binary (default). Returns 9.
MyPos = Instr(1, SearchString, "W")            'Binary compare. Returns 0 ("W" is not found).
```

InStrB

Description	Returns an integer indicating the byte position of the first occurrence of one string within a string containing byte data.
Usage	intVal = **InStrB**([*start*,]*string1*, *string2*[, *compare*])

Arguments

start

Optional. Is any valid non-negative numeric expression that indicates the starting position for each search. Non-integer values are rounded. If omitted, search begins at the first character position. 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. (default)
vbTextCompare	1	Perform a textual comparison

Return

An integer value indicating the byte position.

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 **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. If *start* contains Null, a runtime error occurs. If *start* is larger than the length of *string2* (*start*>**Len**(*string2*)), 0 will be returned.

See Also

InStr, InStrRev

InStrRev

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

Usage intVal = **InStrRev**(*string1*, *string2*[, *start*[, *compare*]])

Arguments *string1*

Required. String expression being searched.

string2

Required. String expression 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 An integer value indicating the position

InStrRev 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 **Note:** The syntax for the **InStrRev** function is not the same as the syntax for the **InStr** function. Note that with UniCode characters, the second byte is usually non-zero (e.g. Asian characters). If *start* is **Null**, a runtime error will occur. If start > **Len(**string2**)**, 0 will be returned.

See Also **InStr, InStrB**

Example 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)   ‘Binary comparison. Returns 9.
MyPos = InstrRev(SearchString, SearchChar, -1, 1)   ‘Textual comparison. Returns 12
MyPos = InstrRev(SearchString, SearchChar, 8)       'Comparison is binary. Returns 0.
```

Int

Description	Returns the integer portion of a number
Usage	intVal = **Int(***number***)**
Arguments	*number* The *number* argument can be any valid numeric expression.
Return	An integer value
Remarks	If *number* contains Null, **Null** is returned. **Int** is internationally aware, meaning the return value is based on the Locale setting on the PC. The data type will be determined from the size of the Integer part. Possible return data types are Integer, Long, Double. 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*)**).**
See also	**Fix, Round, CInt, CLng**

Example	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.

IsArray

Description	Returns a **Variant** subtype **Boolean** value indicating whether a variable is an array
Usage	*result* = **IsArray**(*varname*)
Arguments	*varname* The *varname* argument can be any variable subtype
Returns	**IsArray** returns **True** if the variable is an array; otherwise it returns **False.**
See also	IsDate, IsEmpty, IsNull, IsNumeric, IsObject, and VarType
Example	Dim MyVariable Dim MyArray(2) MyArray(0) = “Sunday” MyArray(1) = “Monday” MyArray(2) = “Tuesday” MyVariable = IsArray (MyArray) ‘ MyVariable contains **True**

IsDate

Description	Returns a Boolean value indicating whether an expression can be converted to a valid date.
Usage	boolVal = **IsDate**(*expression*)
Arguments	*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.
See Also	**CDate, IsArray, IsEmpty, IsNull, IsNumeric, IsObject, VarType**

Example	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.

IsEmpty

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

Usage boolVal = **IsEmpty**(*expression*)

Arguments *expression*

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

Return A **Boolean** value

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. If two or more variables are concatenated in *expression* and one of them is set to **Empty**, the **IsEmpty** function will return **False** since the *expression* is not empty.

See Also **IsArray, IsDate, IsNull, IsNumeric, IsObject, VarType**

Example The following examples 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.
```

IsNull

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

Usage boolVal = **IsNull**(*expression*)

Arguments *expression*

The *expression* argument can be any valid expression.

Return A **Boolean** value

Remarks **IsNull** returns **True** if *expression* evaluates to **Null**, that is, it contains no valid data; otherwise, **IsNull** returns **False**. 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.

You should always use the **IsNull** function when checking for **Null** values since using the normal operators will return **False** even if one variable is **Null**. For example, 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**.

See Also **IsArray, IsDate, IsEmpty, IsNumeric, IsObject, VarType**

Example 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.
```

IsNumeric

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

Usage boolVal = IsNumeric(*expression*)

Arguments *expression*

The *expression* argument can be any valid expression.

Return A **Boolean** value

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, since it is not considered a numeric expression.

See Also **IsArray, IsDate, IsEmpty, IsNull, IsObject, VarType**

Example 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.
```

IsObject

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

Usage boolVal = **IsObject**(*expression*)

Arguments *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**.

See Also **IsArray, IsDate, IsEmpty, IsNull, IsNumeric, VarType**

Example 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.
```

Join

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

Usage strVal = **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.

Return A **String**

Remarks If delimiter is a zero-length string, all items in the list are concatenated with no delimiters. This function is not to be confused with the SQL Join function

See Also Split

Example 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".
```

LBound

Description	Returns the smallest possible subscript for the indicated dimension of an array.
Usage	intVal = **LBound(***arrayname*[**,** *dimension*]**)**
Arguments	*arrayname* Name of the array variable; follows standard variable naming conventions. *dimension* Whole number (integer) 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.
Return	An **Integer** representing the smallest subscript for an array, which in VBScript is always 0 since VBScript arrays are zero-based. Return value will be a Variant subtype **Long**..
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 in VBScript. **LBound** will raise a runtime error if the array has not been initialized.
See also	**Dim**, **ReDim**, **UBound**
Example	Dim MyArray(3) MsgBox **LBound**(MyArray) ‘ Displays 0

LCase

Function	Converts all alpha characters in a string to lowercase.
Usage	strVal = **LCase**(*string*)
Arguments	*string* Any valid string expression..
Remarks	If string contains Null, **Null** is returned. Only uppercase letters are converted to lowercase; all lowercase letters and non-letter characters remain unchanged.
Return	A **String.**
See Also	UCase
Example	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".

Left

Description	Returns a specified number of characters from the left side of a string
Usage	strVal = **Left**(*string, length*)
Arguments	*string* String expression from which the leftmost characters are returned. *length* Numeric expression indicating how many characters to return.
Return	A **String.**
Remarks	If string contains Null, **Null** is returned. If *length* = 0, a zero-length string("") is returned. If *length* is greater than or equal to the number of characters in string, the entire string is returned. To determine the number of characters in string, use the **Len** function.
See Also	**LeftB, Len, LenB, LTrim, Mid, MidB, Right, RTrim, Trim**

Example	Dim myStr, extStr
	myStr = “UpAndDown”
	extStr = Left(myStr, 2) ‘Returns “Up”

LeftB

Description	Returns a specified number of bytes from the left side of a string
Usage	strVal = **Left**(*string, length*)
Arguments	*string* String expression from which the leftmost bytes are returned. *length* Numeric expression indicating how many bytes to return.
Return	A **String.**
Remarks	The **LeftB** function is used with byte data contained in a string instead of character data. If string contains Null, **Null** is returned. If *length* = 0, a zero-length string("") is returned. If *length* is greater than or equal to the number of characters in string, the entire string is returned. To determine the number of characters in string, use the **Len** function.
See Also	**Left, Len, LenB, LTrim, Mid, MidB, Right, RTrim, Trim**
Example	The following example uses the **Left** function to return the first three characters of MyString

Len

Description	Returns the number of characters in a string.
Usage	intVal = **Len**(*string*)
Arguments	*string* Any valid string expression.
Return	An **Integer**
Remarks	If string contains Null, **Null** is returned. The **Len** function is used with character data contained in a string.
See Also	**Left, LeftB, LenB, LTrim, Mid, MidB, Right, RTrim, Trim**
Example	Dim MyString MyString = Len("VBSCRIPT") ' MyString contains 8.

LenB

Description	Returns the number of bytes used to represent a string.
Usage	**LenB**(*string*)
Arguments	*string* Any valid string expression containing byte data.
Return	An **Integer.**
Remarks	If string contains Null, **Null** is returned. 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.
See Also	**Left, LeftB, Len, LTrim, Mid, MidB, Right, RTrim, Trim**

LoadPicture

Description	Returns a picture object.
Usage	objPict = LoadPicture(*picturename*)
Arguments	*picturename* The *picturename* argument is a string expression that indicates the name of the picture file to be loaded.
Return	An object reference to a picture file
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. Once the picture object is loaded, it can be manipulated by other controls (e.g. ActiveX controls). A runtime error occurs if *picturename* does not exist or is not a valid picture file. Use **LoadPicture**(“”) to clear a particular picture. This function is available on 32-bit platforms only.
Example	objPic = LoadPicture (“c:\mypictures\picture1.jpg”)

Log

Description	Returns the natural logarithm of a number.
Usage	realVal = **Log**(*number*)
Arguments	*number* The number argument can be any valid numeric expression greater than 0.
Return	A **Real.**
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)
See also	**Exp**
Example	see below

```
Function Log10(X)                       ‘ Calculate base-10 logarithm
Log10 = Log(X) / Log(10)
End Function
```

LTrim

Description	Returns a copy of a string without leading spaces (**LTrim**), trailing spaces (**RTrim**), or both leading and trailing spaces (**Trim**).
Usage	strVal = **LTrim**(*string*)
Arguments	*string* Required. Any valid string expression.
Return	A **String.**
Remarks	A space “ “ is **Chr**(32). If *string* contains Null, **Null** is returned.
See Also	**Left, LeftB, Len. LenB, Mid, MidB, Right, RTrim, Trim**
Example	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".
```

Mid

Description	Returns a specified number of characters from any position in a string
Usage	strVal = **Mid**(*string*, *start*[, *length*])
Arguments	*string*
	Any valid string expression from which characters are returned.
	start
	Is the starting position in the character string for extracting the characters.
	length
	Optional. Number of characters to return.
Return	A **String**.
Remarks	If *string* contains Null, **Null** is returned. If *start* is greater than the number of characters in the string, **Mid** returns a zero-length string (""). If *length* is 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. To determine the number of characters in string, use the **Len** function.
See Also	**Left, LeftB, Len. LenB, LTrim, MidB, Right, RTrim, Trim**
Example	The following example uses the **Mid** function to return six characters, beginning with the fourth character, in a string:
	Dim MyVar
	MyVar = **Mid(**"VBScript is fun!"**,** 3**,** 6**)** ' MyVar contains "Script".

MidB

Description	Returns a specified number of bytes from any position in a string containing byte data.
Usage	strVal = **Mid**(*string*, *start*[, *length*])
Arguments	*string*
	Any valid string expression containing byte data from which characters are returned.
	start
	Is the starting position in the character string for extracting the bytes.
	length
	Optional. Number of bytes to be returned.
Return	A **String**.
Remarks	If *string* contains Null, **Null** is returned. If *start* is greater than the number of bytes in the string, **MidB** returns a zero-length string (""). If *length* is omitted or if there are fewer than *length* bytes in the text (including the character at start), all bytes from the start position to the end of the string are returned. To determine the number of bytes in string, use the **LenB** function.
See Also	**Left, LeftB, Len. LenB, LTrim, Mid, Right, RTrim, Trim**

Minute

Description	Returns a whole number between 0 and 59, inclusive, representing the minute of the hour.
Usage	invVal = **Minute(***time***)**
Arguments	*time*
	The time argument is any expression that can represent a time.
Return	An **Integer** value
Remarks	A runtime error occurs if time is not a valid time expression. If time contains Null, **Null** is returned.
See Also	**Date, Day, Hour, Month, Now, Second, Weekday, Year**
Example	Dim MyVar
	MyVar = Minute(Now) ‘ Returns the value of the current minute

Month

Description	Returns a whole number between 1 and 12, inclusive, representing the month of the year
Usage	intVal = **Month**(*date*)
Arguments	*date* The *date* argument is any valid expression that can represent a date.
Return	An integer value
Remarks	A runtime error occurs if time is not a valid time expression. If time contains Null, **Null** is returned.
See Also	**Date, Day, Hour, Minute, Now, Second, Weekday, Year**
Example	Dim MyVar MyVar = Month(Now) ‘MyVar contains the number = the current month.

MonthName

Description	Returns a string indicating the specified month.
Usage	strVal = **MonthName(***month*[**,** *abbreviate*]**)**
Arguments	*month* Required. A number between 1 and 12 for each month of the year, beginning in January. 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 (it is spelled out).
Return	A **String.**
Remarks	A runtime error if *month* is outside the valid range (1-12). **MonthName** is internationally aware, meaning that the returned string is localized by the language specified as part of your locale setting.
See Also	**WeekDayName**
Example	Dim MyVar MyVar = MonthName(10, True) ' MyVar contains "Oct".

MsgBox

Description
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.

Usage
intRet = **MsgBox(**prompt[**,** buttons][**,** title][**,** helpfile**,** context]**)**
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. **See examples below for using multiple buttons.**

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.
vbMsgBoxRight	524288	Right align text
vbMsgBoxRtlReading	1048576	On Hebrew and Arabic systems, specifies that text should appear from right to left.
vbMsgBoxSetForeground	65536	Makes the message box in the foreground window

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 Value
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.

See Also **InputBox**

Example
```
Dim MyVar
MyVar = MsgBox ("Hello World!", 65, "MsgBox Example")
' MyVar contains either 1 or 2, depending on which button is clicked.
myResult = MsgBox("Is this OK?", vbYesNo Or vbQuestion Or vbApplicationModal, "Delete File")
```

Now

Desciption Returns the current date and time according to the setting of your computer's system date and time.

Usage dateVal = **Now()**

Arguments None

Remarks The following example uses the **Now** function to return the current date and time:

See Also **Date, Day, Hour, Month, Minute, Second, Weekday, Year**

Example(s)
```
Dim MyVar
MyVar = Now                                    ' MyVar contains the current date and time.
```

Oct

Description Returns a string representing the octal value of a number

Usage strVal = **Oct**(*number*)

Arguments *number*

The number argument is any valid expression.

Return A **String** value

Remarks Returns up to 11 characters. If *number* is not already a whole number, it is rounded to the nearest whole number before being evaluated. 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.

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

See Also **Hex**

Example
```
Dim MyOct MyOct = Oct(4)                       'Returns 4.
MyOct = Oct(8)                                 'Returns 10.
MyOct = Oct(459)                               'Returns 713.
```

Replace

Description: Returns a string in which a specified substring has been replaced with another substring a specified number of times.

Usage: strVal = **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: A **String. 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 occurrences 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 concludes at the end of the *expression* string. It is not a copy of the original string from start to finish

See Also: **Left, LeftB, Len, LenB, LTrim, Mid, MidB, Right, RTrim, Trim**

Example:

```
Dim MyString
Rem  A binary comparison starting at the beginning of the string.
MyString = Replace("XXpXXPXXp", "p", "Y")                    ' Returns "XXYXXPXXY".

Rem  A textual comparison starting at position 3.
MyString = Replace("XXpXXPXXp", "p", "Y", 3, -1, 1)          ' Returns "YXXYXXY".
```

RGB

Description Returns a whole number representing an RGB color value

Usage intVal = **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. A runtime error occurs if any of the arguments cannot be evaluated to a numeric value.

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
```

Example MyColor = RGB(130, 155, 204)

Right

Description Returns length number of characters from the right side of a string

Usage strVal = **Right**(*string**,** length*)

Arguments *string*

String expression from which the characters are extracted from.

length

Numeric expression indicating how many characters to return (extract).

Remarks If *string* contains Null, **Null** is returned. If *length* is 0, a zero-length string("") is returned. If *length* is greater than or equal to the number of characters in string, the entire string is returned. To determine the number of characters in the string, use the **Len** function.

See Also **Left, LeftB, Len, LenB, Mid, MidB, RightB**

Example 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"
```

RightB

Description Returns length number of bytes from the right side of a string

Usage strVal = **Right**(*string**,** length*)

Arguments *string*

String expression from which the bytes are extracted from.

length

Numeric expression indicating how many bytes to return (extract).

Remarks If *string* contains Null, **Null** is returned. If *length* is 0, a zero-length string("") is returned. If *length* is greater than or equal to the number of bytes in the string, the entire string is returned. To determine the number of bytes in the string, use the **LenB** function.

See Also **Left, LeftB, Len, LenB, Mid, MidB, Right**

Rnd

Description	Returns a random number less than 1 but greater than or equal to 0.
Usage	realVal = **Rnd**[(*number*)]
Arguments	*number* Optional. The number argument can be any valid numeric expression.
Result	A **Real** value.
Return Values	A random number less than 1 but greater than 0.

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

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: 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 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.

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.

See also **Randomize**

Example

```
Const UpperBound = 10
Const LowerBound = 1
Dim counter
For counter = 1 to 10                        ‘ Produces 10 numbers between 1-20
    value = Int((UpperBound-LowerBound+1)*Rnd + LowerBound)
    MsgBox “Random Number is = “ & value
Next
```

Round

Description Returns a number rounded to a specified number of decimal places

Usage **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.

Return A **Variant** subtype **Double.** A number rounded to a specified number of decimal places.

Remarks The **Round** function performs *round to even*, which is different from *round to larger*. The return value is the number closest to the value of *expression*, with the appropriate number of decimal places. If *expression* is exactly halfway between two possible rounded values, the function returns the possible rounded value whose rightmost digit is an even number. (In a round to larger function, a number that is halfway between two possible rounded values is always rounded to the larger number.)

See also **Int** and **Fix**

Example

```
Rem Using 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.

Rem How rounding to even works:
Dim var1, var2, var3, var4, var5
var1 = Round(1.5)                     'var1 contains 2
var2 = Round(2.5)                     'var2 contains 2
var3 = Round(3.5)                     'var3 contains 4
var4 = Round(0.985, 2)                'var4 contains 0.98
var5 = Round(0.995, 2)                'var5 contains 1.00
```

RTrim

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

Usage strVal = **RTrim**(*string*)

Arguments *string*

Required. Any valid string expression.

Remarks If *string* contains Null, **Null** is returned.

See Also **Left, LeftB, LTrim, Mid, MidB, Right, RightB, Trim**

Example 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".
```

ScriptEngine

Description	Returns a string representing the scripting language in use
Usage	**ScriptEngine**
Arguments	none
Return Value	A **String.** The **ScriptEngine** function can return the following strings: VBScript Indicates that Microsoft Visual Basic Scripting Edition is the current scripting engine
Remarks	Other 3rd party ActiveX scripting engines can also be returned if they are installed.
See Also	**ScriptEngineBuildVersion, ScriptEngineMajorVersion, ScriptEngineMinorVersion**
Example	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
```

ScriptEngineBuildVersion

Description	Returns the build version number of the scripting engine in use.
Usage	**ScriptEngineBuildVersion**
Arguments	none
Remarks	The return value corresponds directly to the version information contained in the DLL for the scripting language in use.
See Also	**ScriptEngine, ScriptEngineMajorVersion, ScriptEngineMinorVersion**
Example	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
```

ScriptEngineMajorVersion

Description	Returns the major version number of the scripting engine in use.
Usage	**ScriptEngineMajorVersion**
Arguments	none
Remarks	The return value corresponds directly to the version information contained in the DLL for the scripting language in use.
See Also	**ScriptEngine, ScriptEngineBuildVersion, ScriptEngineMinorVersion**
Example	The following example uses the **ScriptEngineMajorVersion** 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
```

ScriptEngineMinorVersion

Description	Returns the minor version number of the scripting engine in use.
Usage	**ScriptEngineMinorVersion**
Arguments	none
Remarks	The return value corresponds directly to the version information contained in the DLL for the scripting language in use.
See Also	**ScriptEngine, ScriptEngineBuildVersion, ScriptEngineMajorVersion**
Example	The following example uses the **ScriptEngineMinorVersion** 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
```

Second

Description	Returns a whole number between 0 and 59, inclusive, representing the second of the minute.
Usage	dateVal = **Second(***time***)**
Arguments	*time* The time argument is any valid expression that can represent a time.
Remarks	A runtime error will occur if time is not a valid time expression. If time contains Null, **Null** is returned.
See Also	**Date, Day, Hour, Minute, Month, Now, Weekday, Year**
Example	Dim MyVar MyVar = Second(Now) ‘ Returns the value of the current second

SetLocale

Description	Sets the current locale ID value
Usage	**SetLocale**(*lcid*)
Arguments	*lcid* The *lcid* cab 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. If lcid is zero, the locale is set to match the current system setting.
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. **This function can be used in conjunction with the IWS run-time translation tool to automatically switch the language displayed**
See Also	**GetLocale**
Example	SetLocale (”en=gb”)

Sgn

Description	Returns the integer indicating the sign of a number
Usage	intVal = Sgn(*number*)
Arguments	*number* The number argument can be any valid numeric expression.
Return	An **Integer.**
Remarks	The sign of the number argument determines the return value of the **Sgn** function.

If *number* is	**Sgn** returns
Greater than zero	1
Equal to zero	0
Less than zero	-1

See also **Abs**

Example

```
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.
```

Sin

Function	Returns the sine of an angle.
Usage	dblVal = **Sin**(*number*)
Arguments	*number* The number argument can be any valid numeric expression that expresses an angle in radians.
Return	Returns a **Variant** subtype **Double** specifying the sine of 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. Pi = 3.14159
See Also	**Atn, Cos, Tan**

Example

```
Dim MyAngle, MyCosecant
MyAngle = 1.3                          ' Define angle in radians.
MyCosecant = 1 / Sin(MyAngle)          ' Calculate cosecant.
```

Space

Description	Returns a string consisting of the specified number of spaces (“ “).
Usage	strVal = **Space**(*number*)
Arguments	*number* The *number* argument is the number of spaces you want in the string.
Return	A **String.**
Remarks	None
See Also	**String**
Example	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.
```

Split

Description	Returns a zero-based, one-dimensional array extracted from the supplied string expression.
Usage	strVal = **Split**(*expression*[**,** *delimiter*[**,** *count*[**,** *compare*]]])
Arguments	*expression* Required. String expression containing substrings and delimiters. *delimiter* Optional. String character used to identify substring limits. *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

Return	A zero-based, one-dimensional array string.
Remarks	If *expression* is a zero-length string, **Split** returns an empty array, that is, an array with no elements and no data. If *delimiter* is 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. The result of the **Split** function cannot be assigned to a variable of **Variant** subtype **Array**, otherwise a runtime error will occur.
See Also	**Join**
Example	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
```

Sqr

Function	Returns the square root of a number.
Usage	val = **Sqr**(*number*)
Arguments	*number* The number argument can be any valid numeric expression greater than or equal to 0.
Return	Returns the square root of a number.

Example

```
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.
```

StrComp

Description	Performs a string comparison and returns the result.
Usage	intVal = **StrComp(***string1***,** *string2*[**,** *compare*]**)**
Arguments	*string1* Required. Any valid string expression. *string2* Required. Any valid string expression. *compare* Optional. Numeric value indicating the comparison method 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: An **Integer.** 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: **Null** is returned if *string1* or *string2* is Null.

See Also: **String**

Example: 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.
```

String

Description	Returns a character string with a substring repeated a specific number of times.
Usage	strVal = **String(***number, character***)**
Arguments	*number* Length of the returned string. *character* Character code specifying the character or string expression whose first character is used to build the return string.
Return	A **String.**
Remarks	If *number* contains Null, **Null** is returned. If *character* contains **Null**, **Null** is returned. 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.
See Also	**Space, StrComp**
Example	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".
```

StrReverse

Description	Returns a string in which the character order of a specified string is reversed.
Usage	**StrReverse**(*string1*)
Arguments	*string1* The *string1* argument is the string whose characters are to be reversed.
Return	A **String.**
Remarks	If *string1* is a zero-length string (""), a zero-length string is returned. If *string1* is **Null**, a runtime error occurs
Example	The following example uses the **StrReverse** function to return a string in reverse order:

```
Dim MyStr
MyStr = StrReverse("VBScript")         ' MyStr contains "tpircSBV".
```

Tan

Description	Returns the tangent of an angle in radians.
Usage	dblVal = **Tan**(*number*)
Arguments	*number* The number argument can be any valid numeric expression that expresses an angle in radians
Return	A **Variant** of subtype **Double.** Specifies the tangent of 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.
See also	**Atn, Cos, Sin**
Example	

```
Dim MyAngle, MyCotangent, MyValue
MyAngle = 1.3                      ' Define angle in radians.
MyCotangent = 1 / Tan(MyAngle)     ' Calculate cotangent.
MyValue = Tan(10.4)                ‘ Returns 1.475667914
MyValue = Tan(0)                   ‘ Returns 0
```

Time

Description	Returns a **Variant** of subtype **Date** indicating the current system time
Usage	dateVal = **Time**()
Arguments	None
Remarks	None
See Also	**Date, Now**
Example	Dim MyTIme MyTime = Time ‘ Return current system time

```
Dim MyTIme
MyTime = Time                                ‘ Return current system time
```

Timer

Description	Returns a Variant of subtype Single indicating the number of seconds that have elapsed since 12:00AM (midnight)
Usage	realVal = **Timer**()
Arguments	none
Remarks	The timer is reset every 24 hours.
Example	

```
Function TimeIt(N)
    Dim StartTime, EndTime
    StartTime = Timer
    For I = 1 To N
    Next
    EndTime = Timer
    TimeIt = EndTime - StartTime
End Function
```

TimeSerial

Description: Returns a **Variant** of subtype **Date** containing the time for a specific hour, minute, and second.

Usage: dateVal = **TimeSerial**(*hour, minute, second*)

Arguments:

hour

Number or valid expression that evaluated to a number between 0 (12:00 A.M.) and 23 (11:00 P.M.).

minute

Number or valid expression that evaluated to a number between 0 and 59.

second

Number or valid expression that evaluated to a number between 0 and 59.

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.

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.

See Also: **Date, DateSerial, DateValue, Day, Month, Now, TimeValue, Weekday, Year**

Example: 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.
```

TimeValue

Description	Returns a **Variant** of subtype **Date** containing the time
Usage	dateVal = TimeValue(time)
Arguments	*time* Time argument is an expression in the range of 0:00:00 to 23:59:59
Remarks	Date information in *time* is not returned. 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. 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.
See Also	**Date, DateSerial, DateValue, Day, Month, Now, TimeValue, Weekday, Year**
Example	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.
```

Trim

Description	Returns a copy of a string without leading spaces (**LTrim**), trailing spaces (**RTrim**), or both leading and trailing spaces (**Trim**).
Usage	strVal = **Trim**(*string*)
Arguments	*string* Required. Any valid string expression.
Return	A **String.**
Remarks	If string contains Null, **Null** is returned.
See Also	**Left, LeftB, Ltrim, Mid, MidB, Right, RightB, RTrim**
Example	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".
```

TypeName

Description — Returns a string that provides **Variant** subtype information about a variable.

Usage — **TypeName**(*varname*)

Arguments — *varname*

The required *varname* argument can be any variable.

Return — A **String.** 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	Uninitialized
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

See Also — **IsArray, IsDate, IsEmpty, IsNull, IsNumeric, IsObject, VarType**

Example — 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()".
```

UBound

Description — Returns the largest available subscript for the indicated dimension of an array.

Usage — IntVal = **UBound(***arrayname[, dimension*]**)**

Arguments — *arrayname*

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

dimension

Optional whole (integer) 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.

Returns — Returns the largest available subscript for the indicated dimension of an array. If the array is empty, -1 is returned. If the array has not been initialized, a runtime error will occur.

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.

See also — **Dim**, **LBound**, **ReDim**

Example

```
Dim A(100,3,4)
Dim B(3)
myVal = UBound(A,1)                     ‘ Return value = 100
myVal = UBound(A,2)                     ‘ Return value = 3
myVal = UBound(A,3)                     ‘ Return value = 4
myVal = UBound(B)                       ‘ Return value = 3
```

UCase

Description	Converts all alpha characters in a string to uppercase and returns the result.
Usage	strVal = **UCase**(*string*)
Arguments	*string* Any valid string expression.
Return	A **String.**
Remarks	If string contains Null, **Null** is returned. Only lowercase letters are converted to uppercase; all uppercase letters and non-letter characters remain unchanged.
See Also	**LCase**
Example	The following example uses the **UCase** function to return an uppercase version of a string Dim MyWord, MyString, LeftString MyWord = UCase("Hello World") ' Returns "HELLO WORLD".

UnEscape

Description	Decodes a string encoded with the Escape function.
Usage	strVal = **UnEscape**(*charstring*)
Arguments	*charstring* Required. Any valid string expression.
Return	A **String** in UniCode format.
Remarks	The **Unescape** function returns a string (in Unicode format) that contains the contents of *charString*. ASCII character set equivalents replace all characters encoded with the %xx hexadecimal form. Characters encoded in %uxxxx format (Unicode characters) are replaced with the Unicode character with hexadecimal encoding xxxx.
See Also	**Escape**

VarType

Description	Returns a value indicating the subtype of a variable.
Usage	**VarType**(*varname*)
Arguments	*varname* The required *varname* argument can be any variable.
Return	An **Integer.** 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

Remarks
These constants are specified by VBScript. As a result, the names can be used anywhere in your code in place of the actual values.

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.

See Also
IsArray, IsDate, IsEmpty, IsNull, IsNumeric, IsObject, TypeName

Example
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.
```

Weekday

Description	Returns a whole number representing the day of the week
Usage	intVal = **Weekday**(*date*, [*firstdayofweek*])
Arguments	*date* Any valid expression that can represent a date. *firstdayofweek* A constant that specifies the first day of the week. If omitted, **vbSunday** is assumed.
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

Return Value — 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 — If date contains Null, **Null** is returned.

See Also — **Date, Day, Month, Now, Year.**

Example

```
Dim MyDate, MyWeekDay
MyDate = #October 19, 1962#                    ' Assign a date.
MyWeekDay = Weekday(MyDate)
Rem    MyWeekDay contains 6 because MyDate represents a Friday.
```

WeekdayName

Description	Returns a **Variant** of subtype **String** indicating the specified day of the week.
Usage	strDayName = **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. Value is between 1 and 7. *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 (is spelled out). *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

Return	A **Variant** of subtype **String** indicating the specified day of the week.
Remarks	A runtime error occurs if *weekday* is outside the valid range of 1-7. **WeekdayName** is internationally aware, which means that the returned strings are localized into the language that is specified in the system's locale settings.
See Also	**MonthName**
Example	

```
Dim MyDate
MyDate = WeekDayName(6, True)          ' MyDate contains Fri.
```

Year

Description	Returns a whole number representing the year
Usage	**Year**(*date*)
Arguments	*date* The *date* argument is any valid expression that can represent a date.
Remarks	If date contains Null, **Null** is returned. A runtime error occurs if *date* is not a valid date expression.
See Also	**Date, Day, Month, Now, Weekday**
Example(s)	

```
Dim MyDate, MyYear
MyDate = #October 19, 1962#            ' Assign a date.
MyYear = Year(MyDate)                  ' MyYear contains 1962.
```

VBScript Derived 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)

VBScript Statements

VBScript Statements			
Call	**ExecuteGlobal**	**Private**	**Select Case**
Class	**Exit**	**Property Get**	**Set**
Const	**For Each…Next**	**Property Let**	**Stop**
Dim	**For…Next**	**Property Set**	**Sub**
Do…Loop	**Function**	**Public**	**While…Wend**
Erase	**If…Then…Else**	**Randomize**	**With**
End	**On Error**	**ReDim**	
Execute	**Option Explicit**	**Rem**	

VBScript Declaration Statements

Function	Description
Class	Declares the name of a class, as well as a definition of the variables, properties, and methods that comprise the class
Const	Declares constants for use in place of literal values
Dim	Declares variables and allocates storage space
Function	Declares the name, arguments, and code that form the body of a **Function** procedure
Option Explicit	Forces explicit declaration of all variables in a script.
Private	Declares private variables and allocates storage space. Declares, in a **Class** block, a private variable.
Property Get	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
Property Let	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
Property Set	Sets a reference to an object
Public	Declares public variables and allocates storage space. Declares, in a **Class** block, a public variable
ReDim	Declare dynamic array variables, and allocates or reallocates storage space at the procedural level
Sub	Declares the name, arguments, and code that form the body of a Sub procedure.

VBScript Array Statements

Function	Description
Dim	Declares variables and allocates storage space
Erase	Reinitializes the elements of fixed-size arrays and deallocates dynamic-array storage space.
ReDim	Declare dynamic array variables, and allocates or reallocates storage space at the procedural level

VBScript Procedure Statements

Function	Description
Call	Transfers control to a Sub or Function procedure
End Function	End of a Function
End Sub	End of a Sub
Exit Function	Exit a Function, generally as a result of a condition
Exit Property	Forces an exit from inside a Property Set function
Exit Sub	Exit a Subroutine, generally as a result of a condition
Function	Declares the name, arguments, and code that form the body of a Function procedure
Sub	Declares the name, arguments, and code that form the body of a Sub procedure (Subroutine).

VBScript Assignment Statements

Function	Description
Set	Assigns an object reference to a variable or property, or associates a procedure reference with an event.

VBScript Comment Statements

Comments	Description
Rem or **‘**	Includes explanatory remarks in a program

VBScript Error Handling Functions

Error Handling	Description
On Error	Enables or disables error-handling

Call

Description Transfers control to a **Sub** or **Function** procedure

Usage **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

Example

```
Function MyFunction(text)
    MsgBox text
End Function

Call MyFunction("Hello World")
MyFunction “Hello World”
```

Class

Description: Declares the name of a class, as well as a definition of the variables, properties, and methods that comprise the class

Usage:
Class *classname*
    statements
End Class

Arguments:
classname
    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. You must instantiate an object to use it, using the **Set** command; i.e. **Set** objname = New classname

See Also: **Property Get, Property Let, Property Set**

Example:

```
Class SignOn
    Private MyName, MyLevel                 ‘Variable declaration
    Public Property Let UsrName(strName)    ‘Set the property value for user name
        MyName = strName
    End Property
    Public Property Let UsrLevel(strLevel)  ‘Set the property value for user level
        MyLevel = strLevel
    End Property
    Public Property Get UsrName             ‘Return the property value
        UsrName = MyName
    End Property
    Public Property Get UsrLevel            ‘Return the property value
        UsrLevel = MyLevel
    End Property
    Public Sub LogOnMsg                     ‘LogOnMsg is a method. No parameters passed
        MsgBox MakeMsg(MyLevel)
    EndSub
    Private Function MakeMsg(strLevel)
        Select Case StrLevel
            Case “User”
                MakeMsg = “Hello “ & MyName & vbCrLf & “Logged on as “ & MyLevel
            Case “Supervisor”
                MakeMsg = “Welcome “ & MyName & vbCrLf & “Your level is “ & MyLevel
        End Select
    End Function
End Class

Dim LogOn
Set LogOn = New SignOn                      ‘Instantiate the object
With LogOn
    .UsrName = “Joe”                        ‘Set the name property
    .UsrLevel = “Supervisor”                ‘Set the level property
    .LogOnMsg                               ‘Invoke logon method
End With
Set LogOn = Nothing
```

Const

Description	Declares constants for use in place of literal values
Usage	[**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.

Example

```
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.
```

Dim

Description	Declares variables and allocates storage space
Usage	Dim *varname*[([*subscripts*])][, <u>*varname*</u>[([*subscripts*])]] . . .

Arguments

varname
: Name of the variable, following 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

Variables declared with the Dim statement at the script level are available to all procedures within the script. Variables declared within a procedure are available only within the procedure. A **Dim** statement with empty parentheses declares a dynamic array, which can be defined later within a procedure using the **ReDim** statement.

Returns

N/A

Example

```
Dim counter                 ‘ Declare a variable
Dim counter1, counter2      ‘ Declares two variables
Dim item(9)                 ‘ Declares an array with 10 elements
Dim item()                  ‘ Declares a dynamic array
```

Do…Loop

Description
Repeats a block of statements while a condition is **True** or until a condition becomes **True**.

Usage

```
Do [{While | Until} condition]
   [statements]
   [Exit Do]
   [statements]
Loop ' or 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.

Example

```
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
```

Erase

Description	Reinitializes the elements of fixed-size arrays and deallocates storage space used if it is a dynamic-array.
Usage	**Erase** *array*
Arguments	*array* The *array* argument is the name of the array variable to be reinitialized or erased
Return	N/A
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 re-declare the array variable's dimensions using a **ReDim** statement.

See also **Dim, ReDim**

Example

```
Dim NumArray(9)              ‘ Declare a fixed-size array
Dim DynamicArray()           ‘ Declare a dynamic array
ReDim DynamicArray(9)        ‘ Allocate storage space
Erase NumArray               ‘ Each element is reinitialized
Erase DynamicArray           ‘ Free memory that was used by array
```

End

Description	Ends a procedure or a block of code
Usage	**End** [**Class | Function | If | Property | Select | Sub | Type | With**]
Arguments	None
Return	N/A
Remarks	Must be used with a procedure statement of a block of code. Provides the normal termination to the procedure or block of code. Must choose the appropriate form of the statement to match the procedure statement or block of code.
See Also	**Exit**

Example

```
If a = b Then
    b = b +2
End If
```

Execute

Description	Executes one or more specified statements in the local namespace.
Usage	**Execute** *statement*
Arguments	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.

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:

Example

```
Sub Proc1                                   'Declare procedure.
    Dim X                                   'Declare X in local scope.
    X = "Local"                             'Assign local X a value.
    Execute "Sub Proc2: MsgBox X: End Sub"  ‘Create a subroutine. Proc2 is local in scope
    MsgBox Eval("X")                        'Print local X.
    Proc2                                   'Invoke Proc2 in Proc1's scope.
End Sub

Rem Main Program
Dim X, s                                    'Declare X in global scope.
X = "Global"                                'Assign global X a value.
Proc2                                       ‘Error - Proc2 is unavailable outside Proc1.
Proc1                                       'Invokes Proc1.
s = “ Main Program”
Execute (“X = X & s”)                       ‘Concatenates strings
Execute "Sub Proc2: MsgBox X: End Sub"
Proc2                                       'Succeeds as Proc2 is now available globally.
```

The result when executing the above code is:

Local	From MsgBox Eval(“X”) in Proc1
Global	From Proc2 statement in Proc1
Global Main Program	From Proc2 statement in Main program

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
```

ExecuteGlobal

Description Executes one or more specified statements in the global namespace.

Usage **Execute** *statement*

Arguments 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.

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:

The difference between Execute and ExecuteGlobal is that Execute operates in the local namespace while ExecuteGlobal operates in the Global namespace. The ExecuteGlobal statement will have limited applicability since IWS does not support a global namespace for variables.

Example

```
Sub Proc1                                    'Declare procedure.
    Dim X                                    'Declare X in local scope.
    X = "Local"                              'Assign local X a value.
    Execute "Sub Proc2: MsgBox X: End Sub"   ‘Create a subroutine. Proc2 is local in scope
    MsgBox Eval("X")                         'Print local X.
    Proc2                                    'Invoke Proc2 in Proc1's scope.
End Sub

Rem Main Program
Dim X, s                                     'Declare X in global scope.
X = "Global"                                 'Assign global X a value.
Proc2                                        ‘Error - Proc2 is unavailable outside Proc1.
Proc1                                        'Invokes Proc1.
s = “ Main Program”
Execute (“X = X & s”)                        ‘Concatenates strings
Execute "Sub Proc2: MsgBox X: End Sub"
Proc2                                        'Succeeds as Proc2 is now available globally.
```

The result when executing the above code is:

Local	From MsgBox Eval(“X”) in Proc1
Global	From Proc2 statement in Proc1
Global Main Program	From Proc2 statement in Main program

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
```

Exit

Description Allows premature exiting of a block of code

Usage **Exit** [**Do | For | Function | Property | Sub**]

Arguments None

Return N/A

Remarks Must be used with a procedure statement of a block of code. Provides early termination. Must choose the appropriate form of the statement to match the procedure statement or block of code.

See Also **End**

Example

```
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
```

For Each…Next

Description Repeats a group of statements for each element in an array or a collection.

Usage

```
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.*

Return N/A

Remarks The **For Each** block is entered if there is at least one element in the array or the collection. 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. 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.

Example

```
Function ShowFileList (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 & vbCrLf
Next
MsgBox “Files in “ & folderspec & “ = “ & s
End
```

For…Next

Description	Repeats a group of statements a specified number of times.

Usage

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

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.

Note that changing the value of the counter while inside a loop can make debugging the code difficult

Example(s)

```
For j = 1 to 10
    For j= 1 to 10
        For I = 1 to 10
            …..
        Next
    Next
Next
```

Function

Description
Declares the name, arguments, and code that form the body of a **Function** procedure

Usage
[**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.
    The *arglist* argument has the following syntax and parts:
        [**ByVal | ByRef**] *varname*[**( )**]
            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.
statements
    Any group of statements to be executed within the body of the **Function** procedure.
    expression

Return
Value of the **Function**.

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.

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**.

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: Function procedures can be recursive, that is, they can call themselves to perform a given task. However, recursion can lead to stack overflow.

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.

See Also **Sub**

Example 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
```

If …Then…Else

Description Conditionally executes a group of statements, depending on the value of an expression.

Usage **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:

1) A numeric or string expression that evaluates to **True** or **False**. If condition is Null, condition is treated as **False**.
2) 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. 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.

Example

```
If A > 10 then A = A + 1
If C = 10 then D = 5 Else E = 4
```

On Error

Description	Enables or disables error handling.
Usage	On Error Resume Next On Error GoTo 0
Arguments	none

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**.

See Also **Err object, Exit**

Example 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.
```

Option Explicit

Description	Forces explicit declaration of all variables in a script.
Usage	**Option Explicit**
Arguments	none

Remarks

If used, the **Option Explicit** statement must appear in a script before any other statements. A compile-time error occurs whenever a variable is encountered that has not been previously declared.

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.

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.

Example 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.
```

Private

Description	Declares private variables and allocates storage space. Declares, in a **Class** block, a private variable.
Usage	Private *varname*[([*subscripts*])][*, varname*[([*subscripts*])]] . . .
Arguments	*varname* Name of the variable, following 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.
Returns	N/A
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 re-declare a dimension for an array variable whose size was explicitly specified in a **Private**, Public, or **Dim** statement, an error occurs. When you use the **Private** statement in a procedure, you generally put the **Private** statement at the beginning of the procedure.

Example

```
Private MyNumber                    ‘ Private Variant variable
Private MyArray(9)                  ‘ Private Array variable
Private MyNumber, MyVar             ‘ Multiple Private declarations
```

Property Get

Description	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.

Usage

```
[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.

See Also
Property Let, Property Set

Example

```
Class myExample
    Private myName
    Public Property Let cName (strName)          'Sets the value
        myName = strName
    End Property
    Public Property Get cName()                  'Returns the value
        cName = myName
    End Property
End Class
```

Property Let

Description	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.

Usage

```
[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.

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

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.

See Also **Property Get, Property Set**

Example

```
Class myExample
    Private myName
    Public Property Let cName (strName)        ‘Sets the value
        myName = strName
    End Property
    Public Property Get cName()                ‘Returns the value
        cName = myName
    End Property
End Class
```

Property Set

Description
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.

Usage
[**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
Property Set is very similar to **Property Let** except that the **Property Set** procedure is used exclusively for object-based properties.

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.

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).

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.

See Also
Property Get, Property Let

Example
```
Class FileHelper
    Private myFSO                           ‘Define a variable to be used for an object
    Public Property Set FSO(objFso)         ‘Set Property
        Set myFSO = objFso                  ‘Defines the object
    End Property
End Class
```

Public

Description	Declares public variables and allocates storage space. Declares, in a **Class** block, a public variable.
Usage	Public *varname*[([*subscripts*])][, *varname*[([*subscripts*])]] . . .
Arguments	*varname* Name of the variable, following 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.
Returns	N/A
Remarks	**Public** statement variables are available to all procedures in all scripts. **Note: This is subject to InduSoft restrictions contained in IWS.** 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 re-declare a dimension for an array variable whose size was explicitly specified in a **Private**, **Public**, or **Dim** statement, an error occurs.
See Also	**Private**

Example

```
Public MyNumber                    ‘Public Variant variable
Public MyArray(9), MyVar           ‘Multiple Public declarations
```

Randomize

Description	Initializes the random number generator.
Usage	Randomize(*number*)
Arguments	*number* The number argument can be any valid numeric expression
Returns	N/A
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. 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
See Also	**Rnd**

Example

```
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                 ‘Print it
    Response = MsgBox (“roll again? “. vbYesNo)
Loop
```

ReDim

Description	Declare dynamic array variables, and allocates or reallocates storage space at the procedural level
Usage	**ReDim** [*Preserve*] *varname*(subscripts) [, *varname*(*subscripts*)]
Arguments	*Preserve* Optional. Preserves the data in an existing array when you change the size of the single dimension or the last dimension (only). If an array is contracted, data in the last elements will still be lost. There is a high overhead associated with using the Preserve functionality and should only be used when necessary. *varname* Required, Name of the array variable, following standard variable naming conventions. Can be any Variant subtype. *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 in VBScript since arrays are zero-based.
Returns	Returns a **Variant** containing an Array
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. Note that if you make an array smaller than it was originally, data in the eliminated elements is lost. A dynamic array must be declared without dimension subscripts.
See also	**Dim, Set**
Example(s)	Dim X() ‘ Declare a dynamic array ReDim X(10,10,10) ‘ Declares dynamic array variables ReDim Preserve X(10,10,15) ‘ Change the size of the last dimension, preserving data

Rem (or) **‘**

Description	Includes explanatory remarks in a program
Usage	Rem *comment* or ‘ *comment*
Arguments	*comment* The comment argument is the text of any comment you want to include. After the **Rem** keyword, a space is required before comment.
Returns	N/A
Remarks	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.
Example	myStr1 = “control” : Rem This is a comment after a statement, separated by a colon myStr2 = “valve” ‘ This is also a comment but here, no colon is needed Rem This is a comment line. No colon is needed

Select Case

Description

Executes one of several groups of statements, depending on the value of an expression

Usage

```
Select Case testexpression
    [Case expressionlist-n
        [statements-n]] . . .
    [Case Else
        [elsestatements-n]]
End Select
```

Arguments

testexpression
    Any numeric or string expression.

expressionlist-n
    Required if **Case** appears. A comma delimited list of one or more expressions.

statements-n
    One or more statements executed if *testexpression* matches any part of *expressionlist*-.

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.

Example

```
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
```

Set

Description	Assigns an object reference to a variable or property, or associates a procedure reference with an event.
Usage	**Set** objectvar = {objectexpression **|** **New** classname **|** **Nothing**} or **Set** object.eventname = **GetRef**(procname)
Arguments	*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. 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 **GetRef** function for information on using **Set** to associate a procedure with an event.
See Also	**GetRef**
Example	Set fso = CreateObject("Scripting.FileSystemObject") Set d = fso.GetDrive(fso.GetDriveName(drvPath)) Set db = CreateObject(ADODB.Connection')

Stop

Description	Suspends execution
Usage	**Stop**
Arguments	None
Remarks	You can place **Stop** statements anywhere in procedures to suspend execution. Using the **Stop** statement is similar to setting a breakpoint in the code. The **Stop** statement suspends execution, but it does not close any files or clear any variables. The **Stop** statement has no effect unless the script is being debugged. **This function does not work in IWS.**
See Also	**Debug object**
Example	see below

```
For i = 1 to 5
    Debug.Write "loop index is " & i
    'Wait for user to resume
    Stop
Next
```

Sub

Description Declares the name, arguments, and code that form the body of a Sub procedure.

Usage

```
[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.

The *arglist* argument has the following syntax and parts:

[**ByVal** | **ByRef**] *varname*[**( )**]

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.

statements

Any group of statements to be executed within the body of the **Sub** procedure.

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.

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: **Sub** procedures can be recursive, that is, they can call themselves to perform a given task. However, recursion can lead to stack overflow.

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.

See Also	**Function**
Example(s)	Sum sqrit(b) b = b * b End Sub

While…Wend

Description	Executes a series of statements as long as a given condition is **True**.
Usage	**While** *condition* [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 that the **Do...Loop** statement provides a more structured and flexible way to perform looping.

Example

```
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
```

With

Description	Executes a series of statements on a single object

Usage

```
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 re-qualifying 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.

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.

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.

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.

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.

Example

```
With MyLabel
    .Height = 2000
    .Width = 2000
    .Caption = "This is MyLabel"
End With
```

VBScript Objects and Collections

Objects and collections	
Class Object	Matches Collection
Debug Object	RegExp Object
Err Object	SubMatches Collection
Match Object	

These Objects and Collections are “built-in” to VBScript and do not rely on any runtime libraries or ActiveX components.

Class

Description
: Declares the name of a class, as well as a definition of the variables, properties, and methods that comprise the class

Usage
: **Class** *classname*
    statements
  End Class

Arguments
: *classname*
  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. You must instantiate an object to use it, using the **Set** command; i.e. **Set** objname = New classname.

  The **Class** block also supports two special subroutines; **Class_Initialize()** and **Class_Terminate()**. Code in the **Class_Initialize()** subroutine executes one time when the Class is instantiated by the statement Set *objName* = New *classname*. Code in the **Class_Terminate()** subroutine executes once when the Class is terminated by the Set *objName* = Nothing statement or when the Class goes out of scope. The **Class_Initialize()** and **Class_Terminate()** subroutines can be Private or Public, but it is recommended to make these Private so that these subroutines may not be called by another code segment.

See Also
: **Property Get, Property Let, Property Set**

Example

```
Class SignOn
    Private MyName, MyLevel                    ‘Variable declaration
Private Sub Class_Initialize()
    ‘Rem Code here executes when Set objName = Class classname statement is executed
End Sub
Private Sub Class_Terminate()
    ‘Rem Code here executes when Set objName = Nothing statement is executed or
    ‘code goes out of scope.
End Sub
Public Property Let UsrName(strName)           ‘Set the property value for user name
        MyName = strName
    End Property
    Public Property Let UsrLevel(strLevel)     ‘Set the property value for user level
        MyLevel = strLevel
    End Property
    Public Property Get UsrName                    'Return the property value
        UsrName = MyName
    End Property
    Public Property Get UsrLevel                   'Return the property value
        UsrLevel = MyLevel
    End Property
    Public Sub LogOnMsg                            'LogOnMsg is a method. No parameters passed
        MsgBox MakeMsg(MyLevel)
    EndSub
    Private Function MakeMsg(strLevel)
        Select Case StrLevel
            Case "User"
                MakeMsg = "Hello " & MyName & vbCrLf & "Logged on as " & MyLevel
            Case "Supervisor"
                MakeMsg = "Welcome " & MyName & vbCrLf & "Your level is " & MyLevel
        End Select
    End Function
End Class

Rem the program starts here
Dim LogOn
Set LogOn = New SignOn                             'Instantiate the object
With LogOn
    .UsrName = "Joe"                               'Set the name property
    .UsrLevel = "Supervisor"                       'Set the level property
    .LogOnMsg                                      'Invoke logon method
End With
Set LogOn = Nothing
```

Debug

NOTE: the Debug object is not currently compatible with IWS. The Debug object is documented for consistency purposes only.

Function	The Debug object is an intrinsic global object that can send an output to a script debugger, such as the Microsoft Script Debugger.
Remarks	The **Debug** object cannot be created directly, but it is always available for use. The **Write** and **WriteLine** methods of the **Debug** object display strings in the Immediate window of the Microsoft Script Debugger at run time. If the script is not being debugged, the methods have no effect.[2]
Method	**Write**
Description	Sends strings to the script debugger
Usage	**Debug.Write** ([*str1* [,*str2* [, …[, *strN*]]]]
Arguments	*str1...strN* Optional. Strings to send to the script debugger
Remarks	The **Write** method sends strings to the Immediate window of the Microsoft Script Debugger at run time. If the script is not being debugged, the **Write** method has no effect. The **Write** method is almost identical to the **WriteLine** method. The only difference is that the **WriteLine** method sends a newline character after the strings are sent.
Example	Dim counter Counter = 30 Debug.Write "The value of counter is " & counter
Method	**WriteLine**
Description	Sends strings to the script debugger, followed by the newline character
Usage	**Debug.WriteLine** ([*str1* [,*str2* [, …[, *strN*]]]]
Arguments	*str1...strN* Optional. Strings to send to the script debugger
Remarks	The **WriteLine** method sends strings to the Immediate window of the Microsoft Script Debugger at run time. If the script is not being debugged, the **WriteLine** method has no effect. The **WriteLine** method is almost identical to the **Write** method. The only difference is that the **Write** method does not send a newline character after the strings are sent.
Example	Dim counter Counter = 30 Debug.Write "The value of counter is " & counter

[2] See http://msdn.microsoft.com for additional information on the Microsoft Script Debugger

Err

Function	Contains information about the last run-time error. Accepts the **Raise** and **Clear** methods for generating and clearing run-time errors.
Usage	val = Err.Property Err.Method
Arguments	Varies with properties and methods used (see below)
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 — VBScript, 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**.

For more information, see Microsoft Web site.[3]

Property **Description**

Function:	Returns or sets a descriptive string associated with an error.
Usage:	**Err.Description** [= *stringexpression*]
Arguments	*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

Example:

```
On Error Resume Next
Error.Raise 39                          'This is a non-defined VBScript error
Err.Description =”Pump OverFlow”        'Define the error message
MsgBox “Error type is “ & Err.Description
```

Property **HelpContext**

Function:	Sets or returns a context ID for a topic in a Help File.
Usage:	**Err.HelpContext** [= *contextID*]
Arguments	*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.

Example:

```
On Error Resume Next
Const usercontextID = 10
Error.Raise 48                          'Error Loading DLL
Err.HelpFile = “myDLL.hlp”              'The help file
Err.HelpContext = usercontextID         'Specify the user context ID
If Err.Number <> 0 Then
    MsgBox “Press F1 for help ” & “Error:” & Error.Description &_
      Err.Helpfile & Err.HelpContext
End If
```

[3] http://www.microsoft.com/technet/scriptcenter/resources/scriptshop/shop1205.mspx
http://www.microsoft.com/technet/scriptcenter/resources/scriptshop/shop0106.mspx

Property	**HelpFile**	
	Function:	Sets or returns a fully qualified path to a Help File
	Usage:	**Err.HelpFile** [= contextID]
	Arguments	*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.

Example

```
On Error Resume Next
Err.Raise 11                            'Divide by 0 error
Err.HelpFile = "myHelpFile.hlp"
Err.HelpContext = usercontextID
If Err.Number <>0 Then
    MsgBox "Press F1 for help" & vbCrLf & "Error: " &Err.Description _
    & Error.HelpFile & Err.HelpContext
End If
```

Property	**Number**	
	Function:	Returns or sets a numeric value specifying an error. **Number** is the **Err** object's default property
	Usage:	**Err. Number** [**=** *errornumber*]
	Arguments	*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**.

Example

```
On Error Resume Next
Err.Raise 11                            'Divide by 0 error
Err.HelpFile = "myHelpFile.hlp"
Err.HelpContext = usercontextID
If Err.Number <>0 Then
    MsgBox "Press F1 for help" & vbCrLf & "Error: " &Err.Description _
    & Error.HelpFile & Err.HelpContext
End If
```

Property	**Source**	
	Function:	Returns or sets the name of the object or application that originally generated the error.
	Usage:	**Err.Source** [**=** *stringexpression*]
	Arguments	*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.

Example

```
On Error Resume Next
Err.Raise 8                         ‘User defined error
Err.Description = “Invalid input”
Err.Source = “MyApplication”
MsgBox “Error Type = “ &Err.Description & “ generated in “ & Err.Source
```

Method **Clear**

Function: Clears all property settings in the **Err** object

Usage: **Err.Clear**

Arguments: none

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

Method **Raise**

Function: Generates a run-time error

Usage: **Err.Raise**(number, source, description, helpfile, helpcontext)

Arguments: *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.

Example

```
On Error Resume Next
Dim Msg
Err.Raise 6                                'Raise an overflow error.
Err.Raise vbObjectError + 1, "SomeObject"  'Raise Object Error #1.
MsgBox ("Error # " & CStr(Err.Number) & " " & Err.Description & Err.Source)
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
Err.Clear                                     ‘ Clear the error
```

Match

Description	Provides access to the read-only properties of a regular expression match.
Usage	For Each **Match** in **Matches** *strRet* = **Match.***prop* Rem other statement can go here Next
Arguments	varies with properties and methods used **Match** The Match object. Does not need to be instantiated **Matches** The Matches collection. Needs to be instantiated in a **Set** statement *prop* A **Match** object property *strRet* Return value.
Return	The return value and type depends on the **Match** property used
See Also	**Length** property, **Value** property, **FirstIndex** property
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.
Example	See example under **Matches** collection

Property	FirstIndex
Description	Returns the position in a search string where a match occurs
Usage	strRet = *objMatch.FirstIndex*
Arguments	None
Return	A numeric value indicating the position in a string where the match occurs.
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).

Property	Length
Description	Returns the length of a match found in a search string.
Usage	strRet = *objMatch.Length*
Arguments	None
Return	A numeric value indicating the length of a match string
Remarks	Always used with the **Match** object

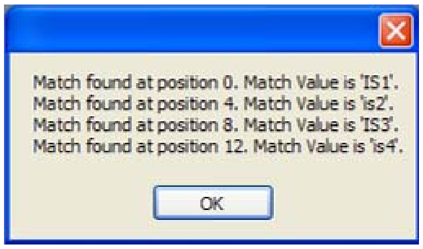
Property	Value
Description	Returns the value or text of a match found in a search string.
Usage	strRet = *objMatch.Length*
Arguments	None
Return	A String containing the match found in the search string
Remarks	Always used with the **Match** object

Matches

Description	Collection of regular expression **Match** objects.
Usage	Set **Matches** = *objRegexp*.Execute(*string)*
Arguments	*objRegexp* A RegExp object that was previously instantiated *string* A command string to execute for the RegExp object
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.
See Also	**RegExp, Match**

Example

```
Dim regEx, Match, Matches, RetStr, srExp           'Create variable.
strExp = “Is1 is2 IS3 is4”                         ‘Input string
Set regEx = New RegExp                             'Instantiate RegExp object
regEx.Pattern =   “is.”                            'Set pattern.
regEx.IgnoreCase = True                            'Set case insensitivity.
regEx.Global = True                                'Set global applicability.
Set Matches = regEx.Execute(strExp)                'Execute search.
RetStr = “”                                        ‘Zero out string
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
MsgBox RetStr
```



```
Match found at position 0. Match Value is 'IS1'.
Match found at position 4. Match Value is 'is2'.
Match found at position 8. Match Value is 'IS3'.
Match found at position 12. Match Value is 'is4'.
```

RegExp

Description	Provides simple regular expression support.	
Usage	Varies with properties and methods used	
Arguments	Varies with properties and methods used	
Remarks		
Property	**Global**	
	Function:	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.
	Usage:	**RegExp.Global** [= **True** | **False**]
	Arguments	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	See example below
Property	**Pattern**	
	Function:	Sets or returns the regular expression pattern being searched for.
	Usage:	**RegExp.Pattern** [= "*searchstring*"]
	Arguments	*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 tables describe the characters that can be used.

Position Matching
Position matching involves the use of the ^ and $ to search for beginning or ending of strings. Setting the pattern property to "^VBScript" will only successfully match "VBScript is cool." But it will fail to match "I like VBScript."

Literals
Literals can be taken to mean alphanumeric characters, ACSII, octal characters, hexadecimal characters, UNICODE, or special escaped characters. Since some characters have special meanings, we must escape them. To match these special characters, we precede them with a "\" in a regular expression.

Character Classes
Character classes enable customized grouping by putting expressions within [] braces. A negated character class may be created by placing ^ as the first character inside the []. Also, a dash can be used to relate a scope of characters. For example, the regular expression "[^a-zA-Z0-9]" matches everything except alphanumeric characters. In addition, some common character sets are bundled as an escape plus a letter.

Repetition
Repetition allows multiple searches on the clause within the regular expression. By using repetition matching, we can specify the number of times an element may be repeated in a regular expression.

Alternation & Grouping
Alternation and grouping is used to develop more complex regular expressions. Using alternation and grouping techniques can create intricate clauses within a regular expression, and offer more flexibility and control.

Back References
Back references enable the programmer to refer back to a portion of the regular expression. This is done by use of parenthesis and the backslash (\) character followed by a single digit. The first parenthesis clause is referred by \1, the second by \2, etc.

Position Matching

Symbol	Function
\	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 "(".
^	Matches the beginning of input.
$	Matches the end of input.
\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".

Literals

Symbol	Function
AlphaNum	Matches alphabetical and numerical characters literally.
\n	Matches a newline character.
\f	Matches a form-feed character.
\r	Matches a carriage return character.
\t	Matches a tab character.
\v	Matches a vertical tab character.
\?	Matches ?
*	Matches *
\+	Matches +
\.	Matches .
\\|	Matches |
\{	Matches {
\}	Matches }
\\	Matches \
\[	Matches [
\]	Matches]
\(	Matches (
\)	Matches)
\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.
\xn	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.

Character Classes

Symbol	Function
[xyz]	Match any one character enclosed in the character set. "[abc]" matches "a" in "plain".
[*a-z***]**	Match any one character in the specified range. For example, "[a-z]" matches any lowercase alphabetic character in the range "a" through "z". "[a-e]" matches "b" in "basketball".
[^xyz]	Match any one character not enclosed in the character set. "[^a-e]" matches "s" in "basketball".
[^*m-z***]**	Matches any character not in the specified range. For example, "[m-z]" matches any character not in the range "m" through "z".
.	Match any single character except \n (newline).
\w	Match any word character including underscore. Equivalent to [a-zA-Z_0-9].
\W	Match any non-word character. Equivalent to [^a-zA-Z_0-9].
\d	Match any digit. Equivalent to [0-9].
\D	Match any non-digit. Equivalent to [^0-9].
\s	Match any space character (e.g. space, tab, form-feed, etc). Equivalent to [\t\r\n\v\f].
\S	Match any non-space character. Equivalent to [^ \t\r\n\v\f].

Repetition

Symbol	Function
{n}	Match exactly n occurrences of a regular expression. n must be a non-negative integer. "\d{5}" matches 5 digits. For example, "o{2}" does not match the "o" in "Bob," but matches the first two o's in "foooood".
(n,}	Match n or more occurrences of a regular expression. n must be a non-negative integer. "\s{2,}" matches at least 2 space characters. 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}	Matches n to m number of occurrences of a regular expression. n and m must be non-negative integers. "\d{2,3}" matches at least 2 but no more than 3 digits. For example, "o{1,3}" matches the first three o's in "fooooood." "o{0,1}" is equivalent to "o?".
?	Match zero or one occurrences. Equivalent to {0,1}. "a\s?b" matches "ab" or "a b" "a?ve?" matches the "ve" in "never"
*	Match zero or more occurrences. Equivalent to {0,}.
+	Match one or more occurrences. Equivalent to {1,}.
\num	Matches num, where num is a positive integer. A reference back to remembered matches. For example, "(.)\1" matches two consecutive identical characters.

Alternation & Grouping

Symbol	Function
()	Grouping a clause to create a clause. May be nested. "(ab)?(c)" matches "abc" or "c".
(pattern)	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 "\(" or "\)".
x | y	Alternation combines clauses into one regular expression and then matches any of the individual clauses; i.e. matches x or y. "(ab)|(cd)|(ef)" matches "ab" or "cd" or "ef".

BackReferences

Symbol	Function
()\n	Matches a clause as numbered by the left parenthesis "(\w+)\s+\1" matches any word that occurs twice in a row, such as "hubba hubba."

	Remarks	See example below
Property	**IgnoreCase**	
	Function:	Sets or returns a **Boolean** value that indicates if a pattern search is case-sensitive or not.
	Usage:	**RegExp.IgnoreCase** [= True | False]
	Arguments	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	See example below
Method	**Execute** Method	
	Function:	Executes a regular expression search against a specified string.
	Usage:	**RegExp.Execute**(*string*)
	Arguments	*string* Required. The text string upon which the regular expression is executed
	Return	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.
	Remarks	The actual pattern for the regular expression search is set using the Pattern property of the RegExp object.
Method	**Replace**	
	Function:	Replaces text found in a regular expression search.
	Usage:	**RegExp.Replace**(*string1, string2*)
	Arguments	*string1* Required. *String1* is the text string in which the text replacement is to occur *string2* Required. *String2* is the replacement text string.
	Return	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.
	Remarks	The actual pattern for the text being replaced is set using the **Pattern** property of the **RegExp** object.

Methods **Test**

Function: Executes a regular expression search against a specified string and returns a **Boolean** value that indicates if a pattern match was found

Usage: **RegExp.Test**(*string*)

Arguments *string*

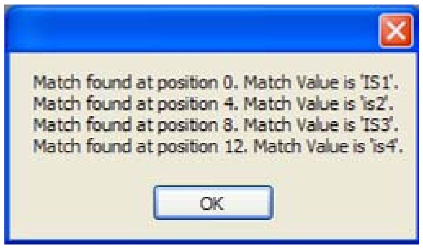
Required. The text string upon which the regular expression is executed

Return The **Test** method returns **True** if a pattern match is found; **False** if no match is found.

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.

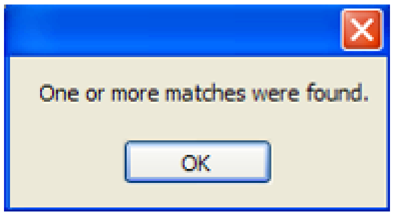
Example

```
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

Rem Program Starts here
MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
```



```
Match found at position 0. Match Value is 'IS1'.
Match found at position 4. Match Value is 'is2'.
Match found at position 8. Match Value is 'IS3'.
Match found at position 12. Match Value is 'is4'.
```

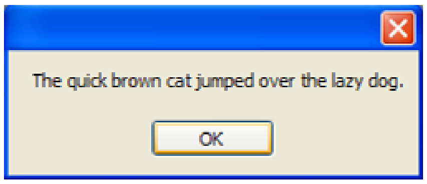
```
Function RegExpTest(patrn, strng)
Dim regEx, retVal                                   ' Create variable.
Set regEx = New RegExp                              ' Create regular expression.
regEx.Pattern = patrn                               ' Set pattern.
regEx.IgnoreCase = False                            ' 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

Rem Program Starts here
MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
```



```
One or more matches were found.
```

The **Replace** method can replace subexpressions in the pattern. The following call to the function ReplaceTest swaps the first pair of words in the original string:

```
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

Rem Program Starts here
MsgBox(ReplaceTest("fox", "cat"))     ' Replace 'fox' with 'cat'.
```



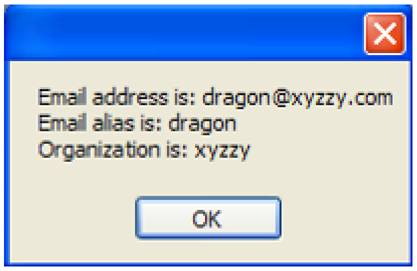
```
The quick brown cat jumped over the lazy dog.
```

SubMatches

Description	A collection of regular expression submatch strings.
Usage	varies with properties and methods used
Arguments	varies with properties and methods used
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.

Example

```
Function SubMatchTest(inpStr)
Dim oRe, oMatch, oMatches
Set oRe = New RegExp
oRe.Pattern = "(\w+)@(\w+)\.(\w+)"              'Look for an e-mail address
Set oMatches = oRe.Execute(inpStr)               'Get the Matches collection
Set oMatch = oMatches(0)                         'Get the first item in the Matches collection
retStr = "Email address is: " & oMatch & vbNewline       'Create the results string.

'The Match object is the entire match - dragon@xyzzy.com
'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

Rem Program Starts here
MsgBox(SubMatchTest("Please send mail to dragon@xyzzy.com. Thanks!"))
```



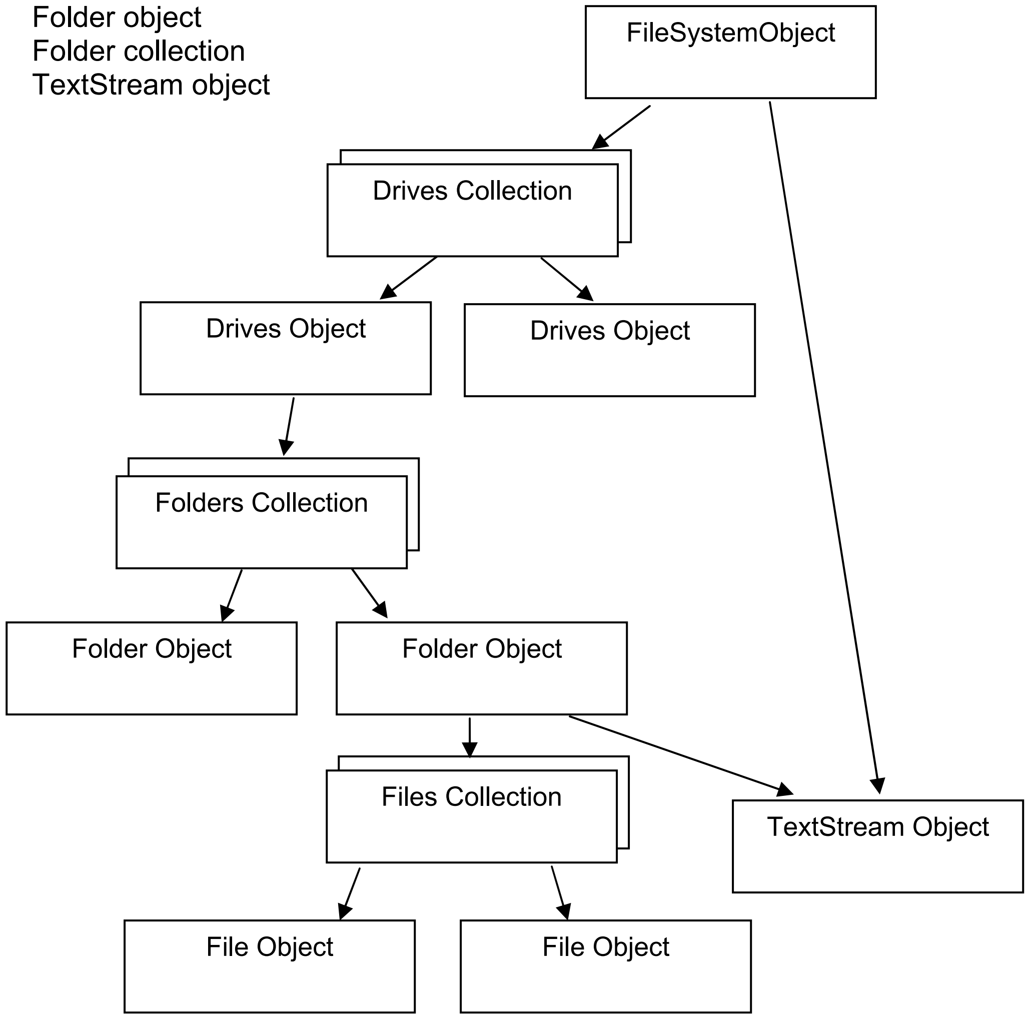
```
Email address is: dragon@xyzzy.com
Email alias is: dragon
Organization is: xyzzy
```

Scripting Type Library

The Scripting Type Library consists of the following item:

- **Dictionary** Object
  The **Dictionary** object is part of the VBScript Scripting Library and is used to store name/value pairs (known as key/item respectively) in an array. The key is a unique identifier for the corresponding item. The key cannot be used of any other item in the same **Dictionary** object. A **Dictionary** object is similar to a normal array, except that instead of using a numeric index, a key is used.

- **FileSystemObject** Object Model
  The **FileSystemObject** is an object model that is part of the VBScript Scripting Library and is used to gain access to a local computer or network share computer file system. It can access drives, folders and files Collections of drives, folders and files can also be retrieved. In addition, the **FileSystemObject** can create, write to and read Text files.
    - Drive object
    - Drives collection
    - File object
    - File collection
    - Folder object
    - Folder collection
    - TextStream object



Dictionary Object

The dictionary object is contained in the Scripting Type library and is a general-purpose data structure that is a cross between a link list and an array. The dictionary stores data and makes the data accessible from one variable. The advantages of a dictionary over an array are:

- You can use “keys” to identify items in the dictionary. Keys are usually strings or integers but can be any data type other than an array or a dictionary. Keys must be unique.
- Methods are provided to add new items and check for existing items in the dictionary
- The dictionary size can be changed without calling the ReDim statement
- Automatically “shifts up” the remaining items when any item in the dictionary is deleted

You can use a **Dictionary** when you need to access random elements frequently or need to access information contained in the array based on its value, not position.

Dictionary

Description Is an associative array that can store any type of data. Data is accessed by a key.
Remarks Keys must be unique.

Property **CompareMode**

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

Arguments: *Compare*
Optional. If provided, compare is a value representing the comparison mode. Values are:
0 = Binary
1 = Text
2 = Database
Values >2 can be used to refer to comparisons using specific Locale IDs (LCID)

Return: Comparison mode

Remarks: An error occurs if you try to change the comparison mode of a Dictionary object that already has data

Use Object.**CompareMode**[= *compare*]

Example:
```
Dim d
Set d = CreateObject(“Scripting.Dictionary”)
d.CompareMode = vbTextCompare
d.Add “a”, “Chicago”
d.Add “b”, “New York”
d.Add “A” = “Austin”                    ‘ Method fails because “b” already exists
```

Property **Count**

Description: Returns the number of items pairs in a **Dictionary** object.

Usage: Object.**Count**

Arguments: None

Return: Integer value of the count of item pairs in a **Dictionary** object.

Remarks: Read Only.

Example:
```
Dim d, item_count
Set d = CreateObject(“Scripting.Dictionary”)
d.Add “a”, “Chicago”
d.Add “b”, “New York”
item_count = d.Count            ‘ Count the items in a Dictionary object
MsgBox item_count
```

Property **Item**

Description: Sets or returns an item for a specified key in a **Dictionary** object.

Usage: Object.**Item** (*key*) [= *newitem*]

Arguments: *key*

Required. Is the key associated with the item being retrieved or added.

newitem

Optional. If provided, new item is the new value associated with the specified key

Return: None

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

Example:

```
Dim d
Set d = CreateObject(“Scripting.Dictionary”)
d.Add “a”, “Chicago”
d.Add “b”, “New York”
MyItem = d.Item(“a”)
MsgBox MyItem                    ‘ Displays Chicago
d.Item(“b”) = “Austin”           ‘ Change item for key “b” to Austin
```

Property **Key**

Description: Sets a key in a **Dictionary** object.

Usage: Object.**Key** (*key*) = *newkey*

Arguments: *key*

Required. Is the key value being changed

newkey

Required. New value that replaces the specified key

Return: None

Remarks: If the key is not found when changing a key, a new key is created and its associated item is left empty.

Example:

```
Dim d
Set d = CreateObject(“Scripting.Dictionary”)
d.Add “a”, “Chicago”
d.Add “b”, “New York”
d.Key(“a”) = “city1”
d.Key(“b”) = “city2”
```

Method **Add**

Description: Adds the name of a dictionary object

Usage: object.**Add** (*key*, *item)*

Arguments: *key*

Required. The key associated with the item being added. Must be unique.

item

Required. This is the item associated with the key being added.

Return: None. Error occurs if the key already exists

Example:

```
Dim d
Set d = CreateObject(“Scripting.Dictionary”)
d.Add “a”, “Chicago”
d.Add “b”, “New York”
```

Method **Exists**

Description: Determine is a specified key exists in the **Dictionary** object

Usage: object.**Exists** (key)

Arguments: *key*

Required. The key value being searched for

Return: **TRUE** if a specified key exists in the Dictionary object, otherwise **FALSE**

Example

```
Dim d, msg
Set d = CreateObject(“Scripting.Dictionary”)
d.Add “a”, “Chicago”
d.Add “b”, “New York”
Msg = “key does not exist”
if d.Exists (“b”) Then msg = “Key exists”
MsgBox (Msg)                     ‘ Indicate if the key exists
```

Method **Items**

Description: Returns an array containing all the existing items in a **Dictionary** object

Usage: Object.**Items** ()

Arguments: None

Return: Array containing all the existing items in the **Dictionary** object

Example:
```
Dim a, d
Set d = CreateObject(“Scripting.Dictionary”)
d.Add “a”, “Chicago”
d.Add “b”, “New York
a = d.Items
For i = 0 to d.Count – 1
     s = s & a(i) & vbCrLf
Next
MsgBox s                              ‘ Display all the items
```

Method **Keys**

Description: Returns an array containing all the existing keys in a **Dictionary** object

Usage: Object.**Keys** ()

Arguments: None

Return: Array containing all the existing keys in the **Dictionary** object

Example:
```
Dim a, d
Set d = CreateObject(“Scripting.Dictionary”)
d.Add “a”, “Chicago”
d.Add “b”, “New York
a = d.Keys
For i = 0 to d.Count – 1
     s = s & a(i) & vbCrLf
Next
MsgBox s                              ‘ Display all the keys
```

Method **Remove**

Description: Removes a key, item pair from a **Dictionary** object

Usage: Object.**Remove** (*key*)

Arguments: *key*

Required. Is the key associated with the key, item pair you want to remove from the **Dictionary** object

Return: None

Example:
```
Dim a, d
Set d = CreateObject(“Scripting.Dictionary”)
d.Add “a”, “Chicago”
d.Add “b”, “New York”
d.Remove(“b”)                         ‘ Removes the “b, New York” key/item pair
```

Method **RemoveAll**

Description: Removes all key, item pairs from a **Dictionary** object

Usage: Object.**RemoveAll**( )

Arguments: None

Return: None

Example:
```
Dim a, d
Set d = CreateObject(“Scripting.Dictionary”)
d.Add “a”, “Chicago”
d.Add “b”, “New York”
d.RemoveAll                           ‘ Removes all key/item pairs
```

FileSystemObject (FSO)

The FileSystemObject (FSO) object model is part of the VBScript Scripting Type library. It is a COM component and is used to manipulate the Windows File System from VBScript. Note that VBScript does not include commands to access files directly, instead the FSO is used.

The FSO consists of collections (Drives Collection, Folders Collection, and Files Collection) that are a grouping of like objects, and individual objects (Drive object, Folder object, File object, and TextStream object). The individual objects are generally derived from a collection or accessed/created directly through the FSO.

The FSO must be instantiated by the following set of statements:

```
Dim objFso                                        'Declare the variable(s)
Set objFso = CreateObject("Scripting.FileSystemObject")   'Instantiate the FileSystemObject
```

FSO Properties and Methods

Property	**Drives**
Description:	Returns a collection of Drive **o**bjects.
Use:	Set objDrive = *fso*.Drives
Arguments:	*fso* Required. The name of a FileSystemObject object previously instantiated.
Return:	An object containing a collection of Drives objects
Remarks:	Returns a collection of Drives **o**bjects available on the local machine, including networked drives mapped to the local machine. Removable media drives do not have to have media inserted to appear in the Drives Collection.

Example:

```
Dim fso, dc, d, strDrvList
Set fso = CreateObject("Scripting.FileSystemObject")   'Instantiate the FSO object
Set dc = fso.Drives                                    'Instantiate the Drives collection object
strDrvList = ""
For each d in dc                                       'Evaluate each drive in the drives
collection
    strDrvList = strDrvList & d.driveLetter & " – "    'Get the Drive letter
    If d.DriveType = 3 Then                            'See if a network drive
        strDrvList = strDrvList & d.ShareName          'Yes
    ElseIf d.IsReady Then                              'No – is a local drive. Check if ready
        strDrvList = strDrvList & d.VolumeName         'Yes – add to list
    End If
    strDrvList = strDrvList & vbCrLf                   'Add a Cr & Lf and then get next drive
Next
MsgBox strDrvList                                      'Display the list of drives
```

Method: **BuildPath**

Description: Appends a name to an existing path

Use: *fso*.**BuildPath**(*path, name*)

Arguments: *fso*

Required. The name of a FileSystemObject object previously instantiated.

path

Required. Existing path to which *name* is appended. Path can be absolute or relative, and need not specify an existing folder

name

Required. Name being appended to the existing path.

Return: None

Remarks: The **BuildPath** method inserts an additional path separator between the existing path and the name, only if necessary. Does not check for a valid path.

Example:

```
Dim fso, path, newpath
Set fso = CreateObject("Scripting.FileSystemObject")
path = $getAppPath()
newpath = fso.BuildPath(path, "SubFolder")
```

Method: **CopyFile**

Description: Copies one or more files from one location to a new location

Use: *fso*.**CopyFile** (*source, destination*[, *overwrite*])

Arguments: *fso*

Required. The name of a FileSystemObject object previously instantiated.

source

Required. A character string file specification, which can include wildcard characters, for one or more files to be copied.

destination

Required. Character string destination where the file or files from *source* are to be copied. Wildcard characters are not allowed in the destination string.

overwrite

Optional. Boolean value that indicates if existing files are to be overwritten. If **True**, files are overwritten; if **False**, they are not. The default is **True**. Note that **CopyFile** will fail if *destination* has the read-only attribute set, regardless of the value of overwrite.

Return: None

Remarks: Wildcard characters can <u>only</u> be used in the last path component of the source argument. If *source* contains wildcard characters or *destination* ends with a path separator (\), it is assumed that *destination* is an existing folder in which to copy matching files. Otherwise, *destination* is assumed to be the name of a file to create. In either case, three things can happen when a file is copied.

- If *destination* does not exist, *source* gets copied. This is the usual case.
- If *destination* is an existing file, an error occurs if overwrite is **False**. Otherwise, an attempt is made to copy *source* over the existing file.
- If *destination* is a directory, an error occurs. (Occurs because the directory doesn’t exist).

An error also occurs if a *source* using wildcard characters doesn't match any files. The **CopyFile** method stops on the first error it encounters. No attempt is made to roll back or undo any changes made before an error occurs.

Example:

```
Const OverWrite = False
Dim fso, srcFiles, destPath
Set fso = CreateObject("Scripting.FileSystemObject")
srcFiles = $getAppPath() & “Alarm\*.*”
destPath = $getAppPath() & “AlarmHistory”
If fso.FolderExists (destPath) = False Then
    fso.CreateFolder (destPath)
End If
fso.CopyFile  srcFiles, destPath
```

Method: **CopyFolder**

Description: Copies a folder to a new location

Use: *fso.***CopyFolder** (*source, destination*[, *overwrite*])

Arguments: *fso*

Required. The name of a FileSystemObject object previously instantiated.

source

Required. A character string folder specification, which can include wildcard characters, for one or more folders to be copied. Wildcard characters can only be used in the last path component of the *source* argument.

destination

Required. Character string destination where the folder and subfolders from *source* are to be copied. Wildcard characters are not allowed in the destination string.

overwrite

Optional. Boolean value that indicates if existing folders are to be overwritten. If **True**, files are overwritten; if **False**, they are not. The default is **True**.

Return: None

Remarks: If *source* contains wildcard characters or *destination* ends with a path separator (\), it is assumed that *destination* is an existing folder in which to copy matching folders and subfolders. Otherwise, *destination* is assumed to be the name of a folder to create. In either case, four things can happen when an individual folder is copied.

- If *destination* does not exist, the source folder and all its contents gets copied. This is the usual case.
- If *destination* is an existing file, an error occurs.
- If *destination* is a directory, an attempt is made to copy the folder and all its contents. If a file contained in *source* already exists in *destination*, an error occurs if overwrite is false. Otherwise, it will attempt to copy the file over the existing file.
- If *destination* is a read-only directory, an error occurs if an attempt is made to copy an existing read-only file into that directory and overwrite is false.

An error also occurs if a *source* using wildcard characters doesn't match any folders. The **CopyFolder** method stops on the first error it encounters. No attempt is made to roll back or undo any changes made before an error occurs

Example:

```
Const OverWrite = False
Dim fso, srcPath, destPath
Set fso = CreateObject("Scripting.FileSystemObject")
srcPath = $getAppPath() & “*”
destPath = fso.GetParentFolderName(srcPath) & “SaveApp”
If fso.FolderExists (destPath) = False Then
    fso.CreateFolder (destPath)
End If
fso.CopyFolder  srcPath, destPath, OverWrite
```

Method: **CreateFolder**

Description: Creates a new folder in the specified location

Use: *fso*.CreateFolder(*foldername*)

Arguments: f*so*

Required. The name of a FileSystemObject object previously instantiated.

foldername

Required. A character string expression that identifies the folder to create.

Return: None

Remarks: An error occurs if the specified folder already exists.

Example:

```
Dim fso, destPath
Set fso = CreateObject("Scripting.FileSystemObject")
destPath = $getAppPath() & “AlarmHistory”
If fso.FolderExists (destPath) = False Then
    fso.CreateFolder (destPath)
End If
```

Method:	**CreateTextFile**
Description:	Creates a specified file name and returns a **TextStream** object that can be used to read from or write to the file
Use:	Set objfile = *fso.***CreateTextFile**(*filename*[, *overwrite*[, *Unicode*]])
Arguments:	*fso* Required. The name of a FileSystemObject object previously instantiated *filename* Required. A string expression that identifies the file to create *overwrite* Optional. Boolean value that indicates whether you can overwrite an existing file. The value is **True** if the file can be overwritten, **False** if it can't be overwritten. If omitted, existing files are not overwritten (default **False**). *unicode* Optional. Boolean value that indicates whether the file is created as a Unicode or ASCII file. If the value is **True,** the file is created as a Unicode file. If the value is **False,** the file is created as an ASCII file. If omitted, an ASCII file is assumed.
Remarks:	None
Example:	Dim fso, myFile Set fso = CreateObject("Scripting.FileSystemObject") Set myFile = fso.CreateTextFile("c:\testfile.txt", True, False) myFile.WriteLine("This is a test.") myFile.Close Set Myfile = Nothing Set fso = Nothing

Method:	**DeleteFile**
Description:	Deletes a specified file
Use:	*fso*.DeleteFile (*filename[, force]*)
Arguments:	*fso* Required. The name of a FileSystemObject object previously instantiated *filename* Required. The name of the file to delete. The filename can contain wildcard characters in the last path component. *force* Optional. Boolean value that is **True** of files with the read-only attribute set are to be deleted; **False** if they are not. **False** is the default.
Return:	None
Remarks:	An error occurs if no matching files are found. The **DeleteFile** method stops on the first error it encounters. No attempt is made to roll back or undo any changes that were made before an error occurred.
Example:	Dim fso, myFile Set fso = CreateObject("Scripting.FileSystemObject") myFile = “C:\TempData\Log*.dat” fso.DeleteFile(myFile) Set fso = Nothing

Method: **DeleteFolder**

Description: Deletes the specified folder and its contents

Use: *fso*.DeleteFolder (*folderspec[, force]*)

Arguments: *fso*

Required. The name of a FileSystemObject object previously instantiated

folderspec

Required. The name of the folder to delete. The folderspec can contain wildcard characters in the last path component.

force

Optional. Boolean value that is **True** of folders with the read-only attribute set are to be deleted; **False** if they are not. **False** is the default.

Return: None

Remarks: The **DeleteFolder** method does not distinguish between folders that have contents and those that do not. The specified folder is deleted regardless of whether or not it has contents. An error occurs if no matching folders are found. The **DeleteFolder** method stops on the first error it encounters. No attempt is made to roll back or undo any changes that were made before an error occurred.

Example:

```
Dim fso, myFolder
Set fso = CreateObject("Scripting.FileSystemObject")
myFolder = “C:\TempData\”
fso.DeleteFolder(myFolder)
Set fso = Nothing
```

Method: **DriveExists**

Description: Determines whether or not a specified drive exists

Use: *fso*.DriveExists (*drivespec*)

Arguments: *fso*

Required. The name of a FileSystemObject object previously instantiated

drivespec

Required. A drive letter or a complete path specification.

Return: Returns a boolean **True** if the specified drives exists, otherwise returns **False.**

Remarks: For drives with removable media, the **DriveExists** method returns **true** even if there are no media present. Use the **IsReady** property of the **Drive** object to determine if a drive is ready.

Example:

```
Dim fso, drv, msg
Set fso = CreateObject("Scripting.FileSystemObject")
drv = “e:\”
If fso.DriveExists(drv) Then
    msg = "Drive " & UCase(drv) & " exists."
Else
    msg = "Drive " & UCase(drv) & " doesn't exist."
End If
MsgBox msg
```

Method:	**FileExists**
Description:	Determines whether or not a specified file exists
Use:	*fso*.FileExists (*filespec*)
Arguments:	*fso* Required. The name of a FileSystemObject object previously instantiated *filespec* Required. The name of the file whose existence is to be determined. A complete path specification (either absolute or relative) must be provided if the file isn't expected to exist in the current folder
Return:	Returns a boolean **True** if the specified file exists, otherwise returns **False.**
Remarks:	None

Example:

```
Dim fso, myFile, msg
Set fso = CreateObject("Scripting.FileSystemObject")
myFile = $getAppPath() & “data\Mydata.mdb”
If fso.FileExists(myFile) Then
     msg = myFile & " exists."
Else
     msg = myFile & "doesn't exist."
End If
MsgBox msg
```

Method:	**FolderExists**
Description:	Determines whether or not a specified folder exists
Use:	*fso*.FolderExists (*folderspec*)
Arguments:	*fso* Required. The name of a FileSystemObject object previously instantiated *folderspec* Required. The name of the folder whose existence is to be determined. A complete path specification (either absolute or relative) must be provided if the folder isn't expected to exist in the current folder
Return:	Returns a boolean **True** if the specified folder exists, otherwise returns **False.**
Remarks:	None

Example:

```
Dim fso, myFolder, msg
Set fso = CreateObject("Scripting.FileSystemObject")
myFolder = $getAppPath() & “data\”
If fso.FolderExists(myFolder) Then
     msg = myFolder & " exists."
Else
     msg = myFolder & "doesn't exist."
End If
MsgBox msg
```

Method: **GetAbsolutePathName**

Description: Returns a complete and unambiguous path name that cannot be easily determined from the specified path information.

Use: strPath = *fso*.GetAbsolutePathName(*pathspec*)

Arguments: *fso*

Required. The name of a FileSystemObject object previously instantiated

pathspec

Required. Path specification to change to a complete and unambiguous path

Return: String containing a complete and unambiguous path name

Remarks: A path is complete and unambiguous if it provides a complete reference from the root of the specified drive. A complete path can only end with a path separator character (\) if it specifies the root folder of a mapped drive. Assuming the current directory is c:\mydocuments\reports, the following table illustrates the behavior of the **GetAbsolutePathName** method:

pathspec	Returned path
"c:"	"c:\mydocuments\reports"
"c:.."	"c:\mydocuments"
"c:\"	"c:\"
"c:*.*\may97"	"c:\mydocuments\reports\*.*\may97"
"region1"	"c:\mydocuments\reports\region1"
"c:\..\..\mydocuments"	"c:\mydocuments"

Example:

```
Dim fso, pathSpec, myPath
Set fso = CreateObject("Scripting.FileSystemObject"      ‘Current directory is
c:\mydocuments\reports
pathSpec = “C:\”
myPath = fso.GetAbsolutePathName(pathSpec)              ‘Returns c:\mydocuments\reports
```

Method: **GetBaseName**

Description: Returns just the name of the object specified. It removes all other information including the extension

Use: strBaseName = *fso*.GetBaseName(*path*)

Arguments: *fso*

Required. The name of a FileSystemObject object previously instantiated

path

Required. The path specification for the component whose base name is to be returned.

Return: String containing the name of the object specified.

Remarks: The **GetBaseName** method works only on the provided path string. It does not attempt to resolve the path, nor does it check for the existence of the specified path. The **GetBaseName** method returns a zero-length string (“”) if no component matches the *path* argument.

Example:

```
Dim fso, filespec, baseName
Set fso = CreateObject("Scripting.FileSystemObject"
filespec = $getAppPath() & “recipes.xml”
baseName = fso.GetBaseName (filespec)                   ‘Returns “recipes”
```

Method: **GetDrive**
Description: Returns a **Drive** object corresponding to the drive for a specified path
Use: objDrv = *fso*.GetDrive(*drivespec*)
Arguments: *fso*
    Required. The name of a FileSystemObject object previously instantiated
drivespec
    Required. The *drivespec* argument can be a drive letter (c), a drive letter with a colon appended (c:), a drive letter with a colon and path separator appended (c:\), or any network share specification (\\computer2\share1).
Return: Drive Object corresponding to the drive for a specified path
Remarks: For network shares, a check is made to ensure that the share exists. An error occurs if drivespec does not conform to one of the accepted forms or does not exist.
Example:

```
Dim fso, drvPath, d, s
Set fso = CreateObject("Scripting.FileSystemObject")
drvPath = "c:"
Set d = fso.GetDrive(fso.GetDriveName(drvPath))
s = "Drive " & UCase(drvPath) & " - "
s = s & d.VolumeName   & vbCrLf
s = s & "Free Space: " & FormatNumber(d.FreeSpace/1024, 0)
s = s & " Kbytes"
MsgBox s
```

Method: **GetDriveName**
Description: Returns a string containing the name of the drive for a specified path
Use: strName = *fso*.GetDriveName(*path*)
Arguments: *fso*
    Required. The name of a FileSystemObject object previously instantiated
path
    Required. The path specification for the component whose drive name is to be returned.
Return: String containing the name of the drive for a specified path
Remarks: The **GetDriveName** method works only on the provided path string. It does not attempt to resolve the path, nor does it check for the existence of the specified path. The **GetDriveName** method returns a zero-length string ("") if the drive can't be determined.
Example:

```
Dim fso, drvPath, GetAName
Set fso = CreateObject("Scripting.FileSystemObject")
drvPath = "c:"
GetAName = fso.GetDriveName(drvPath)                'Returns "c:"
```

Method: **GetExtensionName**
Description: Returns a string containing the extension name for the last component in a path.
Use: strExtName = *fso*.GetExtensionName(*path*)
Arguments: *fso*
    Required. The name of a FileSystemObject object previously instantiated
path
    Required. The path specification for the component whose drive name is to be returned.
Return: String containing the extension name for the last component in a path.
Remarks: For network drives, the root directory (\) is considered to be a component. The **GetExtensionName** method returns a zero-length string ("") if no component matches the path argument.
Example:

```
Dim fso, drvPath, ExtName
Set fso = CreateObject("Scripting.FileSystemObject")
drvPath = $getAppPath() & "recipes.xml"
ExtName = fso.GetExtensionName(drvPath)             'Returns "xml"
```

Method: **GetFile**

Description: Returns a **File** object corresponding to the file in the specified path. The file object methods and properties can be accessed. See ***File Object*** for the file object's methods and properties.

Use: objFile = *fso*.GetFile(*fileSpec*)

Arguments: *fso*

Required. The name of a FileSystemObject object previously instantiated

fileSpec

Required. The filespec is the path (absolute or relative) to a specific file.

Return: File Object

Remarks: An error occurs if the specified file does not exist. The **GetFile** method does not support the use of wildcard characters, such as ? or *.

Example:

```
Dim fso, fileSpec, f, s
Set fso = CreateObject("Scripting.FileSystemObject")
fileSpec = $getAppPath() & “recipes.xml”
Set f = fso.GetFile(fileSpec)
s = f.Path & vbCrLf
s = s & "Created: " & f.DateCreated & vbCrLf
s = s & "Last Accessed: " & f.DateLastAccessed & vbCrLf
s = s & "Last Modified: " & f.DateLastModified
MsgBox s
```

Method: **GetFileName**

Description: Returns the last component of a specified path (file name or folder name) that is not part of the drive specification.

Use: strName = *fso.*GetFileName(*fileSpec*)

Arguments: *fso*

Required. The name of a FileSystemObject object previously instantiated

fileSpec

Required. The path (absolute or relative) to a specific file.

Return: String containing the last component of a specified path

Remarks: The **GetFileName** method works only on the provided path string. It does not attempt to resolve the path, nor does it check for the existence of the specified path. The **GetFileName** method returns a zero-length string (“”) if *pathspec* does not end with the named component.

Example:

```
Dim fso, fileSpec, s
Set fso = CreateObject("Scripting.FileSystemObject")
fileSpec = $getAppPath() & “recipes.xml”
s = fso.GetFile(fileSpec)                              ‘Returns “recipes.xml”
MsgBox s
```

Method: **GetFileVersion**
Description: Returns the version number of a specified file
Use: strVersionNum = *fso*.GetFileVersion(*fileSpec*)
Arguments: *fso*

Required. The name of a FileSystemObject object previously instantiated

fileSpec

Required. The path (absolute or relative) to a specific file.

Return: String containing the version number of a specified file
Remarks: The **GetFileVersion** method works only on the provided path string. It does not attempt to resolve the path, nor does it check for the existence of the specified path. The **GetFileVersion** method returns a zero-length string (“”) if *pathspec* does not end with the named component.
Example:

```
Dim fso, fileSpec, s
Set fso = CreateObject("Scripting.FileSystemObject")
fileSpec = “c:\windows\system32\notepad.exe”
s = fso.GetFile(fileSpec)                         ‘Returns “5.1.2600.2180”
If Len(s) Then
    MsgBox “File Version is : “ & s
Else
    MsgBox “No File Version information is available”
End If
```

Method: **GetFolder**
Description: Returns a **Folder** object corresponding to the folder in a specified path
Use: objFolder = *fso*.**GetFolder**(*folderSpec*)
Arguments: *fso*

Required. The name of a FileSystemObject object previously instantiated

folderSpec

Required. The folderspec is the path (absolute or relative) to a specific folder.

Return: Returns a folder object
Remarks: Since this method creates an object, you need to use it with the Set command. An error occurs if the specified folder does not exist.
Example:

```
Dim fso, drvPath, f, fc, s
Set fso = CreateObject("Scripting.FileSystemObject")
drvPath = $getAppPath()
Set f = fso.GetFolder(drvPath)
Set fc = f.SubFolders
s = “”
For Each x in fc
    s = s & x.Name & vbCrLf
Next
MsgBox s                                  ‘Displays a list of folders in the App directory
```

Method:	**GetParentFolderName**
Description:	Returns a string containing the name of the parent folder of the last component in the specified path
Use:	strName = *fso*.GetParentFolderName(*path*)
Arguments:	*fso* Required. The name of a FileSystemObject object previously instantiated *path* Required. The path specification for the component whose parent folder name is to be returned.
Return:	String containing the name of the parent folder
Remarks:	The **GetParentFolderName** method works only on the provided path string. It does not attempt to resolve the path, nor does it check for the existence of the specified path. The **GetParentFolderName** method returns a zero-length string (“”) if there is no parent folder for the component specified in the *path* argument.
Example:	Dim fso, drvPath, s Set fso = CreateObject("Scripting.FileSystemObject") drvPath = $getAppPath() s = fso.GetParentFolderName(drvPath) MsgBox “Parent Folder = “ & s ‘Returns “c:\My Documents\InduSoft Web Studio v6.1 Projects”

Method:	**GetSpecialFolder**
Description:	Returns the special folder specified
Use:	strFolderName = *fso*.GetSpecialFolder(*folderSpec*)
Arguments:	*fso* Required. The name of a FileSystemObject object previously instantiated *folderSpec* Required. Then name of the special folder to be returned. Can be any of the following constants:

Constant	Value	Description
WindowsFolder	0	The Windows folder containing files installed by the Windows operating system
SystemFolder	1	The (Windows) System folder containing libraries, fonts and device drivers
TemporaryFolder	2	The Temp folder is used to store temporary files. Its path is found in the TMP environment variable.

Return:	String containing the name of the parent folder
Remarks:	None

Example:	Dim fso, WinFolder, SysFolder
	Set fso = CreateObject("Scripting.FileSystemObject")
	WinFolder = fso.GetSpecialFolder(0) & “\” ‘Result is “C:\Windows\”
	SysFolder = fso.GetSpecialFolder(1) & “\” ‘Result is “C:\Windows\system32\”

Method:	**GetStandardStream**
Description:	Returns a **TextStream** object corresponding to the standard input, output, or error stream

Note:

- The **GetStandardStream** Method does not work with IWS and if you use it, you will get an error. **GetStandardStream** only works for standard I/O when CScript is the VBScript Interpreter. For operator I/O, use MsgBox and InputBox instead.

Method: **GetTempName**

Description: Returns a randomly generated temporary file or folder name that is useful for performing operations that require a temporary file or folder

Use: strName = *fso.*GetTempName

Arguments: *fso*

Required. The name of a FileSystemObject object previously instantiated

Return: String that contains a randomly generated temporary file or folder name. A random name with a .tmp extension will be returned.

Remarks: The **GetTempName** method does not create a file. It only provides only a temporary file name that can be used with **CreateTextFile** to create a file.

Example:

```
Dim fso, tempFile

Function CreateTempFile
    Const TemporaryFolder = 2
    Dim tfolder, tname, tfile
    Set tfolder = fso.GetSpecialFolder(TemporaryFolder)
    tname = fso.GetTempName
    Set tfile = tfolder.CreateTextFile(tname)
    Set CreateTempFile = tfile
End Function

Set fso = CreateObject("Scripting.FileSystemObject")
Set tempFile = CreateTempFile
tempFile.WriteLine "Hello World"
tempFile.Close
```

Method: **MoveFile**

Description: Moves one or more files from one location to another

Use: *fso*.MoveFile (*source, destination*)

Arguments: *fso*

Required. The name of a FileSystemObject object previously instantiated

source

Required. The path to the file or files to be moved. The source argument string can contain wildcard characters in the last path component only.

destination

Required. The path where the file or files are to be moved. The destination argument can't contain wildcard characters.

Return: None

Remarks: If *source* contains wildcards or *destination* ends with a path separator (\), it is assumed that *destination* specifies an existing folder in which to move the matching files. Otherwise, *destination* is assumed to be the name of a destination file to create. In either case, three things can happen when an individual file is moved:

- If *destination* does not exist, the file gets moved. This is the usual case.
- If *destination* is an existing file, an error occurs.
- If *destination* is a directory, an error occurs.

An error also occurs if a wildcard character that is used in source doesn't match any files. The **MoveFile** method stops on the first error it encounters. No attempt is made to roll back any changes made before the error occurs

Example:

```
Dim fso, drvPath
Set fso = CreateObject("Scripting.FileSystemObject")
drvPath = $getAppPath() & “recipes.xml”
fso.MoveFile drvPath, "c:\backup\"
```

Method:	**MoveFolder**
Description:	Moves one or more folders from one location to another.
Use:	*fso*.MoveFolder (*source, destination*)

Arguments: *fso*

Required. The name of a FileSystemObject object previously instantiated

source

Required. The path to the folder or folders to be moved. The source argument string can contain wildcard characters in the last path component only.

destination

Required. The path where the folder or folders are to be moved. The destination argument can't contain wildcard characters.

Return: None

Remarks: If *source* contains wildcards or *destination* ends with a path separator (\), it is assumed that *destination* specifies an existing folder in which to move the matching folders. Otherwise, *destination* is assumed to be the name of a destination folder to create. In either case, three things can happen when an individual folder is moved:

- If *destination* does not exist, the folder gets moved. This is the usual case.
- If *destination* is an existing file, an error occurs.
- If *destination* is a directory, an error occurs.

An error also occurs if a wildcard character that is used in source doesn't match any folders. The **MoveFolder** method stops on the first error it encounters. No attempt is made to roll back any changes made before the error occurs

Example:

```
Dim fso, drvPath
Set fso = CreateObject("Scripting.FileSystemObject")
drvPath = $getAppPath()
fso.MoveFolder drvPath, "c:\backup\"
```

Method:	**OpenTextFile**
Description:	Opens a specified file and returns a **TextStream** object that can be used to read from, write to, or append to a file.
Use:	oTSO = *fso*.OpenTextFile(*filename* [*, iomode[, create[, format]]*])

Arguments: *fso*

Required. The name of a FileSystemObject object previously instantiated

filename

Required. A string expression that identifies the file to open.

iomode

Optional. Indicates the file input/output mode. Can be one of three constants:

Constant	Value	Description
ForReading	1	Open a file for reading only. You can’t write to this file
ForWriting	2	Open a file for reading & writing
ForAppending	8	Open a file and write to the end of the file

create

Optional. Boolean value that indicates whether a new file can be created if the specified *filename* doesn't exist. The value is **True** if a new file is to be created if it doesn’t exist, **False** if it isn't to be created if it doesn’t exist. If omitted, a new file isn't created (default = **FALSE**).

format

Optional. One of three **Tristate** values used to indicate the format of the opened file. If omitted, the file is opened as ASCII.

Constant	Value	Description
TristateUseDefault	-2	Opens the file using the system default
TristateTrue	-1	Opens the file as Unicode
TristateFalse	0	Opens the file as ASCII

Return: A **TextStream** object

Example:

```
Const ForReading=1, ForWriting=2, ForAppending=8
Dim fso, f
Set fso = CreateObject("Scripting.FileSystemObject")
Set f = fso.OpenTextFile("c:\testfile.txt", ForWriting, True)
f.Write "Hello world!"
f.Close
```

Drives Collection

FSO Property **Drives**

Description: Returns a collection of Drives **o**bjects.

Use: Set objDrives = *fso*.Drives

Arguments: *fso*

Required. The name of a FileSystemObject object previously instantiated.

objDrives

Required. The name of a Drive Collection.

Return: An object containing a collection of Drives objects

Remarks: Returns a collection of Drives **o**bjects available on the local machine, including networked drives mapped to the local machine. Removable media drives do not have to have media inserted to appear in the Drives Collection.

Example:

```
Dim fso, dc, d, strDrvList
Set fso = CreateObject("Scripting.FileSystemObject")   'Instantiate the FSO object
Set dc = fso.Drives                                     'Instantiate the Drives collection object
strDrvList = ""
For each d in dc                                        'Evaluate each drive in the drives
collection
    strDrvList = strDrvList & d.driveLetter & " – "     'Get the Drive letter
    If d.DriveType = 3 Then                             'See if a network drive
        strDrvList = strDrvList & d.ShareName           'Yes
    ElseIf d.IsReady Then                               'No – is a local drive. Check if ready
        strDrvList = strDrvList & d.VolumeName          'Yes – add to list
    End If
    strDrvList = strDrvList & vbCrLf                    'Add a Cr & Lf and then get next drive
Next
MsgBox strDrvList                                       'Display the list of drives
```

Property **Count**

Description: Returns the number of items in a collection. Read only.

Use: intCount = *objDrives*.Count

Arguments: *objDrives*

Required. The name of a Drive Collection previously instantiated.

Return: The number of items in a collection.

Remarks: Read only.

Example:

```
Dim fso, dc, totDrives
Set fso = CreateObject("Scripting.FileSystemObject")   'Instantiate the FSO object
Set dc = fso.Drives                                     'Instantiate the Drives collection object
totDrives = dc.Count
MsgBox "There are " & totDrives & " drives available"
```

Property **Item**

Description: Returns an item (a Drive Name) based on the specified key.

Use: strName = *objDrives*.Item(*key*)

Arguments: *objDrives*

Required. The name of a Drive Collection previously instantiated.

key

Required. The *key* is associated with the *item* being retrieved.

Return: The drive name for a specified key.

Remarks: Read only. This is a function more commonly used with the VBScript dictionary object. (Scripting.Dictionary). The "Item" is similar to a numerical-based index in an array, except that an Item can be character based and it must be unique.

Example:

```
Dim fso, dc, myItem
Set fso = CreateObject("Scripting.FileSystemObject")   'Instantiate the FSO object
Set dc = fso.Drives                                     'Instantiate the Drives collection object
myItem = dc.Item ("c")
MsgBox myItem                                           'Displays "c:"
```

Folders Collection

FSO Method **GetFolder**

Description: Returns a **Folder** object corresponding to the folder in a specified path

Use: objFolder = *fso*.**GetFolder**(*folderspec*)

Arguments: *fso*

Required. The name of a FileSystemObject object previously instantiated

folderspec

Required. The folderspec is the path (absolute or relative) to a specific folder.

Return: Returns a folder object

Remarks: Since this method creates an object, you need to use it with the Set command. An error occurs if the specified folder does not exist.

Example:

```
Dim fso, drvPath, f, fc, nf
Set fso = CreateObject("Scripting.FileSystemObject")
drvPath = $getAppPath()
Set f = fso.GetFolder(drvPath)              ‘Instantiate the parent folder object
Set fc = f.SubFolders                       ‘Return the subfolder Folders collection
s = “”
For Each x in fc
    s = s & x.Name & vbCrLf                 ‘Iterate through the Folders collection object
Next
MsgBox s                                    ‘Displays a list of subfolders in the App directory
```

Property **Count**

Description: Returns the number of items in a collection. Read only.

Use: intCount = *objFolders*.Count

Arguments: *objFolders*

Required. The name of a Folders Collection previously instantiated.

Return: The number of items in a collection.

Remarks: Read only.

Example:

```
Dim drvPath, fso, fc, f, numf
Set fso = CreateObject(“Scripting.FileSystemObject”)     ‘Instantiate the FSO object
drvPath = $getAppPath()
Set f = fso.GetFolder(drvPath)                           ‘Instantiate the parent folder object
Set fc = f.SubFolders                                    ‘Return the subfolder Folders collection
numf = fc.Count
MsgBox “There are “ & numf & “ subfolders”
```

Property **Item**

Description: Returns an item (a Drive Name) based on the specified key.

Use: strName = *objFolders*.Item(*key*)

Arguments: *objFolders*

Required. The name of a Folders Collection.

key

Required. The *key* is associated with the *item* being retrieved.

Return: The drive name for a specified key.

Remarks: Read only. This is a function more commonly used with the VBScript dictionary object. (Scripting.Dictionary). The “Item” is similar to a numerical-based index in an array, except that an Item can be character based and it must be unique.

Example:

```
Dim drvPath, fso, fc, myItem
Set fso = CreateObject(“Scripting.FileSystemObject”)     ‘Instantiate the FSO object
drvPath = $getAppPath()
Set f = fso.GetFolder(drvPath)                           ‘Instantiate the parent folder object
Set fc = f.SubFolders                                    ‘Return the subfolder Folders collection
myItem = fc.Item (“Web”)
MsgBox myItem                                            ‘displays the entire path to the Web
subfolder
```

Method	**Add**
Description:	Adds a new folder to the Folders collection.
Use:	*objFolders*.Add(*folderName*)
Arguments:	*objFolders* Required. The name of a Folders Collection previously instantiated. *folderName* Required. The name of the new Folder being added.
Return:	None
Remarks:	Adds a subfolder to the parent folder. An error occurs if the *folderName* already exists.

Example:

```
Dim drvPath, fso, fc, numf
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
drvPath = $getAppPath()
Set f = fso.GetFolder(drvPath)                          'Instantiate the parent folder object
Set fc = f.SubFolders                                   'Return the subfolder Folders collection
numf = fc.Count
MsgBox "There are " & numf & " subfolders"              'Returns "7"
fc.Add ("TempData")                                     'Add a "TempData" subfolder
numf = fc.Count
MsgBox "There are " & numf & " subfolders"              'Returns "8"
```

Files Collection

FSO Method **GetFolder**

Description: Returns a **Folder** object corresponding to the folder in a specified path

Use: objFolder = *fso*.**GetFolder**(*folderspec*)

Arguments: *fso*

Required. The name of a FileSystemObject object previously instantiated

folderspec

Required. The folderspec is the path (absolute or relative) to a specific folder.

Return: Returns a folder object

Remarks: Since this method creates an object, you need to use it with the Set command. An error occurs if the specified folder does not exist.

Example:

```
Dim fso, drvPath, f, fc, x, s
Set fso = CreateObject("Scripting.FileSystemObject")
drvPath = $getAppPath() & “Hst”
Set f = fso.GetFolder(drvPath)                 ‘Instantiate the folder object
Set fc = f.Files                               ‘Return the Files collection
s = “”
For Each x in fc
    s = s & x.Name & vbCrLf                    ‘Iterate through the Files collection object
Next
MsgBox s                                       ‘Displays a list of files in the “Hst” subfolder
```

Property **Count**

Description: Returns the number of items in a collection. Read only.

Use: intCount = *objFiles*.Count

Arguments: *objFiles*

Required. The name of a Files Collection object previously instantiated.

Return: The number of items in a collection.

Remarks: Read only.

Example:

```
Dim drvPath, fso, fc, numf
Set fso = CreateObject(“Scripting.FileSystemObject”)    ‘Instantiate the FSO object
drvPath = $getAppPath()
Set f = fso.GetFolder(drvPath)                          ‘Instantiate the parent folder object
Set fc = f.Files                                        ‘Return the subfolder Folders collection
numf = fc.Count
MsgBox “There are “ & numf & “ files”
```

Property **Item**

Description: Returns an item (a Drive Name) based on the specified key.

Use: strName = *objFiles*.Item(*key*)

Arguments: *objFiles*

Required. The name of a Folders Collection object previously instantiated.

key

Required. The *key* is associated with the *item* being retrieved.

Return: The drive name for a specified key.

Remarks: Read only. This is a function more commonly used with the VBScript dictionary object. (Scripting.Dictionary). The “Item” is similar to a numerical-based index in an array, except that an Item can be character based and it must be unique.

Example:

```
Dim drvPath, fso, fc, myItem
Set fso = CreateObject(“Scripting.FileSystemObject”)    ‘Instantiate the FSO object
drvPath = $getAppPath()
Set f = fso.GetFolder(drvPath)                          ‘Instantiate the parent folder object
Set fc = f.Files                                        ‘Return the subfolder Folders collection
myItem = fc.Item (“myApp.app”)
MsgBox myItem                                           ‘displays the entire path to myApp.app
```

Drive Object

The Drive Object lets the programmer refer to a specific disk drive or network share drive. Once the Drive object is instantiated, it can be referred to as an object from VBScript and its various Properties accessed.

The Drive Object is instantiated as follows:

```
Dim fso, d, driveSpec
Set fso = CreateObject("Scripting.FileSystemObject")          'Instantiate the FSO Object
driveSpec = "c"
Set d = fso.GetDrive(driveSpec)                               'Instantiate the Drive Object
```

See the **GetDrive** method under the FileSystemObject Object Model section for additional details on instantiation of the Drive Object.

The Drive object has no Methods, only Properties. These properties are generally read-only and follow the format:

return = *objDrive.Property*
where
return = return value or a returned object
objDrive = the required Drive object ("d" in the examples below)
Property = the Drive object property being accessed

Property	**AvailableSpace**
Description:	Returns the amount of space available to a user on the specified drive or network share drive.
Use:	intSpace = *objDrive.*AvailableSpace
Arguments:	*objDrive* Required. The name of a Drive Object previously instantiated.
Return:	The read-only value returned by the **AvailableSpace** property is typically the same as that returned by the **FreeSpace** property. Differences may occur between the two for computer systems that support quotas.
Remarks:	Read only.
Example:	see below

```
Dim fso, d
Set fso = CreateObject("Scripting.FileSystemObject")      'Instantiate the FSO object
Set d = fso.GetDrive(fso.GetDriveName("c:")
MsgBox "Available Space = " & FormatNumber(d.AvailableSpace/1024, 0) & " Kbytes"
```

Property	**DriveLetter**
Description:	Returns the drive letter of a physical local drive or a network share.
Use:	strLetter = *objDrive.*DriveLetter
Arguments:	*objDrive* Required. The name of a Drive Object previously instantiated.
Return:	The **DriveLetter** property returns a zero-length string ("") if the specified drive is not associated with a drive letter, for example, a network share that has not been mapped to a drive letter.
Remarks:	Read only.
Example:	see below

```
Dim fso, d
Set fso = CreateObject("Scripting.FileSystemObject")      'Instantiate the FSO object
Set d = fso.GetDrive(fso.GetDriveName("c:")
MsgBox "Drive " & d.DriveLetter & ":"
```

Property	**DriveType**
Description:	Returns a value indicating the type of a specified drive.
Use:	intType = *objDrive.*DriveType
Arguments:	*objDrive*
	Required. The name of a Drive Object previously instantiated.
Return:	The **DriveType** property a value indication the type of the specified drive. Return values are:
	0 – unknown
	1 – Removable
	2 – Fixed
	3 – Network
	4 – CD-ROM
	5 – RAM Disk
Remarks:	Read only.
Example:	

```
Dim fso, d, t
Set fso = CreateObject("Scripting.FileSystemObject")      'Instantiate the FSO object
Set d = fso.GetDrive(fso.GetDriveName("c:")
Select Case d.DriveType
     Case 0: t = "Unknown"
     Case 1: t = "Removable"
     Case 2: t = "Fixed"
     Case 3: t = "Network"
     Case 4: t = "CD-ROM"
     Case 5: t = "RAM Disk"
 End Select
 MsgBox "Drive " & d.DriveLetter & ": - " & " is a " & t & " drive"
```

Property	**FileSystem**
Description:	Returns the type of file system in use for the specified drive.
Use:	strType = *objDrive.*FileSystem
Arguments:	*objDrive*
	Required. The name of a Drive Object previously instantiated.
Return:	Available return types include FAT, NTFS, and CDFS.
Remarks:	Read only.
Example:	

```
Dim fso, d
Set fso = CreateObject("Scripting.FileSystemObject")      'Instantiate the FSO object
Set d = fso.GetDrive(fso.GetDriveName("c:")
MsgBox "Drive " & d.DriveLetter & "  Files System type =" & d.FileSystem
```

Property	**FreeSpace**
Description:	Returns the amount of space available to a user on the specified drive or network share drive.
Use:	intSpace = *objDrive.*FreeSpace
Arguments:	*objDrive*
	Required. The name of a Drive Object previously instantiated.
Return:	The read-only value returned by the **FreeSpace** property is typically the same as that returned by the **AvailableSpace** property. Differences may occur between the two for computer systems that support quotas.
Remarks:	Read only.
Example:	

```
Dim fso, d
Set fso = CreateObject("Scripting.FileSystemObject")      'Instantiate the FSO object
Set d = fso.GetDrive(fso.GetDriveName("c:")
MsgBox "Free Space = " & d.FreeSpace/1024 & " Kbytes"
```

Property **IsReady**

Description: Indicates whether the specified drive is ready or not

Use: boolReady = *objDrive.*IsReady

Arguments: *objDrive*

Required. The name of a Drive Object previously instantiated.

Return: Returns **True** if the specified drive is ready; **False** if it is not.

Remarks: Read only.

Example:

```
Dim fso, d, s
Set fso = CreateObject("Scripting.FileSystemObject")     'Instantiate the FSO object
Set d = fso.GetDrive(fso.GetDriveName("c:")
s = "Drive " & d.DriveLetter
If d.IsReady Then
    s = s & " Drive is Ready."
Else
    s = s & " Drive is not Ready."
End If
MsgBox s
```

Property **Path**

Description: Returns the path for a specified drive.

Use: strPath = *objDrive.*Path

Arguments: *objDrive*

Required. The name of a Drive Object previously instantiated.

Return: The path for a specified drive

Remarks: For drive letters, the root drive is not included. For example, the path for the C drive is C:, not C:\.

Example:

```
Dim fso, d
Set fso = CreateObject("Scripting.FileSystemObject")     'Instantiate the FSO object
Set d = fso.GetDrive(fso.GetDriveName("c:"))
MsgBox "Path = " & UCase(d.Path)                         'Returns c:
```

Property **RootFolder**

Description: Returns a **Folder** object representing the root folder of a specified drive.

Use: objFolder = *objDrive.*RootFolder

Arguments: *objDrive*

Required. The name of a Drive Object previously instantiated.

Return: The path for a specified drive

Remarks: Read-only. All the files and folders contained on the drive can be accessed using the returned **Folder** object.

Example:

```
Dim fso, d
Set fso = CreateObject("Scripting.FileSystemObject")     'Instantiate the FSO object
Set d = fso.GetDrive("c:")
MsgBox "RootFolder = " & d.RootFolder                    'Returns "c:\"
```

Property **SerialNumber**

Description: Returns the decimal serial number used to uniquely identify a disk volume.

Use: intSerNum = *objDrive.*SerialNumber

Arguments: *objDrive*

Required. The name of a Drive Object previously instantiated.

Return: A decimal serial number that uniquely identifies a disk volume

Remarks: Read-only. You can use the **SerialNumber** property to ensure that the correct disk is inserted in a drive with removable media.

Example:

```
Dim fso, d
Set fso = CreateObject("Scripting.FileSystemObject")     'Instantiate the FSO object
Set d = fso.GetDrive("c:")
MsgBox "Drive Serial Number = " & d.SerialNumber         'Returns "c:\"
```

Property **ShareName**

Description: Returns the network share name for a specified drive.

Use: strName = *objDrive.*ShareName

Arguments: *objDrive*

Required. The name of a Drive Object previously instantiated.

Return: A string that is the network share name for a specified drive.

Remarks: Read-only. If *object* is not a network drive, the **ShareName** property returns a zero-length string ("").

Example:

```
Dim fso, dc, d
Set fso = CreateObject("Scripting.FileSystemObject")     'Instantiate the FSO object
Set d = fso.GetDrive("c:")
If d.DriveType = 3 Then                                   'See if a network drive
    MsgBox "Network Shared Drive Name = " & d.ShareName
Else
    MsgBox "Not a Network Shared Drive"
End If
```

Property **TotalSize**

Description: Returns the total space, in bytes, of a drive or network shared drive.

Use: intSize = *objDrive.*TotalSize

Arguments: *objDrive*

Required. The name of a Drive Object previously instantiated.

Return: Integer. The total space, in bytes, of a drive or network shared drive

Remarks: Read-only.

Example:

```
Dim fso, d
Set fso = CreateObject("Scripting.FileSystemObject")     'Instantiate the FSO object
Set d = fso.GetDrive("c:")
MsgBox "Total Drive Size = " & d.TotalSize & " bytes"     'Returns the total size of the drive
```

Property **VolumeName**

Description: Sets or returns the volume name of the specified drive. Read/write.

Use: *strName* = *objDrive.*VolumeName

*objDrive.*VolumeName [= *newname*]

Arguments: *objDrive*

Required. The name of a Drive Object previously instantiated..

newname

Optional. If provided, *newname* is the new name of the specified object

Return: String. The volume name of the specified drive.

Remarks: Read/Write.

Example:

```
Dim fso, d
Set fso = CreateObject("Scripting.FileSystemObject")     'Instantiate the FSO object
Set d = fso.GetDrive("c:")
MsgBox "Total Drive Size = " & d.TotalSize & " bytes"     'Returns the total size of the drive
```

Folder Object

The Folder Object allows the programmer refer to a specific folder. Once the Folder object is instantiated, it can be referred to as an object from VBScript and its various Methods and Properties accessed.

The Folder Object is instantiated as follows:

```
Dim fso, f, myPath
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO Object
myPath = $GetAppPath() & "Hst"                          'Define the path to the folder of
interest
Set f = fso.GetFolder(myPath)                           'Instantiate the Drive Object
```

See the **GetFolder** method under the FileSystemObject Object Model section for additional details on instantiation of the Folder Object.

Method **Copy**

Description: Copies a specified folder from one location to another.

Use: *objFolder.*Copy (*destination,* [*overwrite*])

Arguments:
objFolder
Required. The name of a Folder Object previously instantiated.
destination
Required. Destination where the folder is to be copied. Wildcard characters are not allowed.
overwrite
Optional. Boolean value that is **True** (default) if existing folders are to be overwritten, **False** if they are not.

Return: None

Remarks: The results of the **Copy** method on a **Folder** are identical to operations performed using **FileSystemObject.CopyFolder** where the folder referred to by *object* is passed as an argument. You should note, however, that the alternative method is capable of copying multiple folders.

Example:

```
Dim fso, f, myFolder
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFolder = $getAppPath() & "Hst"                        'Application Folder for Historical files
Set f = fso.GetFolder (myFolder)
f.Copy (myFolder & "Temp")                              'Creates folder /HstTemp and copies files
```

Method: **CreateTextFile**

Description: Creates a specified file name and returns a **TextStream** object that can be used to read from or write to the file

Use: Set objFile = *objFolde.***CreateTextFile**(*filename*[, *overwrite*[, *Unicode*]])

Arguments:
objFolder
Required. The name of a Folder Object previously instantiated.
filename
Required. A string expression that identifies the file to create
overwrite
Optional. Boolean value that indicates whether you can overwrite an existing file. The value is **True** if the file can be overwritten, **False** if it can't be overwritten. If omitted, existing files are not overwritten (default **False**).
unicode
Optional. Boolean value that indicates whether the file is created as a Unicode or ASCII file. If the value is **True,** the file is created as a Unicode file. If the value is **False,** the file is created as an ASCII file. If omitted, an ASCII file is assumed.

Remarks: None

Example:

```
Dim fso, myFile
Set fso = CreateObject("Scripting.FileSystemObject")
Set myFile = fso.CreateTextFile("c:\testfile.txt", True, False)
myFile.WriteLine("This is a test.")
myFile.Close
```

Method:	**Delete**
Description:	Deletes a specified folder
Use:	*objFolder.*Delete (*force*)
Arguments:	*objFolder* Required. The name of a Folder Object previously instantiated. *force* Optional. Boolean value that is **True** if folders with the read-only attribute set are to be deleted; **False** if they are not (default).
Return:	None
Remarks:	An error occurs if the specified folder does not exist. The results of the **Delete** method on a **Folder** are identical to operations performed using **FileSystemObject.DeleteFolder**. The **Delete** method does not distinguish between folders that have content and those that do not. The specified folder is deleted regardless of whether or not it has content.
Example:	see below

```
Dim fso, f, myFolder
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFolder = $getAppPath() & "HstTemp"                    'Specify the HstTemp folder in app
directory
Set f = fso.GetFolder (myFolder)
f.Delete                                                'Delete it
```

Method:	**Move**
Description:	Moves a specified folder from one location to another.
Use:	*objFolder.*Move (*destination*)
Arguments:	*objFolder* Required. The name of a Folder Object previously instantiated. *destination* Required. Destination where the folder is to be moved. Wildcard characters are not allowed.
Return:	None
Remarks:	The results of the **Move** method on a **Folder** is identical to operations performed using **FileSystemObject.MoveFolder**. You should note, however, that the alternative methods are capable of moving multiple folders.
Example:	see below

```
Dim fso, f, myFolder
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFolder = $getAppPath() & "HstTemp"                    'Specify the HstTemp folder in app
directory
Set f = fso.GetFolder (myFolder)
f.move("c:\archive")                                    'Move it into c:\archive folder
```

Property: **Attributes**

Description: Sets or returns the attributes of files or folders.

Use: *objFolder.*Attributes = newAttributes
intAttribute = *objFolder*.Attributes

Arguments: *objFolder*
Required. The name of a Folder Object previously instantiated.
newAttributes
Optional. If provided, newAttributes is the new value for the attributes of the specified *object*. The newattributes argument can have any of the following values or any logical combination of the following values:

Constant	Value	Description
Normal	0	Normal file. No Attributes are set.
ReadOnly	1	Read-only file. Attribute is read/write.
Hidden	2	Hidden file. Attribute is read/write.
System	4	System file. Attribute is read/write.
Volume	8	Disk drive volume label. Attribute is read-only
Directory	16	Folder or directory. Attribute is read-only.
Archive	32	File has changed since last backup. Attribute is read/write
Alias	1024	Link or shortcut. Attribute is read-only
Compressed	2048	Compressed file. Attribute is read-only.

Return: Can return an attribute of a file or folder

Remarks: Read/write or read-only, depending on the attribute. The newAttribute can have any valid combination of the above values.

Example:
```
Dim fso, f, attrVal, myFolder
Set fso = CreateObject(“Scripting.FileSystemObject”)    ‘Instantiate the FSO object
myFolder = $getAppPath()                                ‘Specify the app directory
Set f = fso.GetFolder (myFolder)
attrVal = f.Attributes
attrVal = attrVal And 16                                ‘See if a folder
If attrVal = 16 Then
    MsgBox “Object is a folder”
Else
    MsgBox “Object is not a folder”
End If
```

Property: **DateCreated**

Description: Returns the date and time that the specified folder was created.

Use: *objFolder.*DateCreated

Arguments: *objFolder*
Required. The name of a Folder Object previously instantiated.

Return: None

Remarks: Read-only.

Example:
```
Dim fso, f, myFolder
Set fso = CreateObject(“Scripting.FileSystemObject”)    ‘Instantiate the FSO object
myFolder = $getAppPath()                                ‘Specify the app directory
Set f = fso.GetFolder (myFolder)
MsgBox “App Directory created on “ & f.DateCreated
```

Property:	**DateLastAccessed**
Description:	Returns the date and time that the specified folder was last accessed
Use:	*objFolder.*DateLastAccessed
Arguments:	*objFolder*
	Required. The name of a Folder Object previously instantiated.
Return:	None
Remarks:	Read-only.
Example:	Dim fso, f, myFolder

```
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFolder = $getAppPath()                                'Specify the app directory
Set f = fso.GetFolder (myFolder)
MsgBox "App Directory was last accessed on " & f.DateLastAccessed
```

Property:	**DateLastModified**
Description:	Returns the date and time that the specified folder was last modified
Use:	*objFolder.*DateLastModified
Arguments:	*objFolder*
	Required. The name of a Folder Object previously instantiated.
Return:	None
Remarks:	Read-only.
Example:	Dim fso, f, myFolder

```
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFolder = $getAppPath()                                'Specify the app directory
Set f = fso.GetFolder (myFolder)
MsgBox "App Directory was last modified on " & f.DateLastModified
```

Property:	**Drive**
Description:	Returns the drive letter of the drive on which the specified folder resides
Use:	*objFolder.*Drive
Arguments:	*objFolder*
	Required. The name of a Folder Object previously instantiated.
Return:	None
Remarks:	Read-only.
Example:	Dim fso, f, myFolder

```
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFolder = $getAppPath()                                'Specify the app directory
Set f = fso.GetFolder (myFolder)
MsgBox "App Directory is installed on drive  " & f.Drive    'Installed on drive c:
```

Property:	**Files**
Description:	Returns a Files collection consisting of all File objects contained in the specified folder.
Use:	*objFolder.*Files
Arguments:	*objFolder*
	Required. The name of a Folder Object previously instantiated.
Return:	A file collection.
Remarks:	Includes files with hidden and system file attributes set.
Example:	Dim fso, f, fc, myFolder

```
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFolder = $getAppPath()                                'Specify the app directory
Set f = fso.GetFolder (myFolder)
fc = f.files                                            'Return file collection of files in app folder
```

Property:	**IsRootFolder**
Description:	Tests to see if the specified folder is the root folder.
Use:	boolValue = *objFolder.*IsRootFolder
Arguments:	*objFolder* Required. The name of a Folder Object previously instantiated.
Return:	**True** if the specified folder is the root folder; **False** if not.
Remarks:	Includes files with hidden and system file attributes set.

Example:

```
Dim fso, f, n, s, myFolder
Set fso = CreateObject("Scripting.FileSystemObject")     'Instantiate the FSO object
myFolder = $getAppPath()                                 'Specify the app directory
Set f = fso.GetFolder (myFolder)
n = 0
If f.IsRootfolder Then
    MsgBox "The app folder is the root folder"
Else
    s = myFolder & vbCrLf
    Do Until f.IsRootFolder
        Set f = f.ParentFolder
        n = n+1
        s = s & "parent folder is " & f.Name & vbCrLf
    Loop
    MsgBox "Folder was nested " & n & " levels" & vbCrLf & s
End If
```

Property:	**Name**
Description:	Sets or returns the name of a specified folder
Use:	*objFolder.*Name = *newName* strName = *objFolder.*Name
Arguments:	*objFolder* Required. The name of a Folder Object previously instantiated. *newName* Optional. If provided, *newName* is the new name of the specified folder object
Return:	The name of the specified folder.
Remarks:	Read/write.

Example:

```
Dim fso, f, myFolder
Set fso = CreateObject("Scripting.FileSystemObject")     'Instantiate the FSO object
myFolder = $getAppPath()                                 'Specify the app directory
Set f = fso.GetFolder (myFolder)
MsgBox "folder name is " & f.Name                        'Returns the folder name
```

Property: **ParentFolder**
Description: Returns the folder object for the parent of the specified folder
Use: objParent = *objFolder.*ParentFolder
Arguments: *objFolder*
Required. The name of a Folder Object previously instantiated.
Return: The folder object for the parent of the specified folder.
Remarks: Read-only
Example:

```
Dim fso, f, pf, myFolder
Set fso = CreateObject(“Scripting.FileSystemObject”)    ‘Instantiate the FSO object
myFolder = $getAppPath()                                ‘Specify the app directory
Set f = fso.GetFolder (myFolder)
Set pf = f.ParentFolder                                 ‘Get the parent folder
MsgBox “Parent Folder name = “ & pf.Name
```

Property **Path**
Description: Returns the path for a specified folder
Use: strPath = *objFolder.*Path
Arguments: *objFolder*
Required. The name of a Folder Object previously instantiated.
Return: The path for a specified folder
Remarks: None
Example:

```
Dim fso, f, myFolder
Set fso = CreateObject(“Scripting.FileSystemObject”)    ‘Instantiate the FSO object
myFolder = $getAppPath()                                ‘Specify the app directory
Set f = fso.GetFolder (myFolder)
MsgBox “Path = “ & UCase(f.Path)                        ‘Display path to app folder
```

Property **ShortName**
Description: Returns the short name used by programs that require the earlier 8.3 naming convention.
Use: strName = *objFolder.*ShortName
Arguments: *objFolder*
Required. The name of a Folder Object previously instantiated.
Return: The short name for the folder object
Remarks: None
Example:

```
Dim fso, f, myFolder
Set fso = CreateObject(“Scripting.FileSystemObject”)    ‘Instantiate the FSO object
myFolder = $getAppPath()                                ‘Specify the app directory
Set f = fso.GetFolder (myFolder)
MsgBox “Short name = “ & f.ShortName                    ‘Display short name of app folder
```

Property **ShortPath**
Description: Returns the short path used by programs that require the earlier 8.3 naming convention.
Use: strPath = *objFolder.*ShortPath
Arguments: *objFolder*
Required. The name of a Folder Object previously instantiated.
Return: The short path for the folder object
Remarks: None
Example:

```
Dim fso, f, myFolder
Set fso = CreateObject(“Scripting.FileSystemObject”)    ‘Instantiate the FSO object
myFolder = $getAppPath()                                ‘Specify the app directory
Set f = fso.GetFolder (myFolder)
MsgBox “Short pathname = “ & f.ShortPath                ‘Display short path of app folder
```

Property	**Size**
Description:	Returns the size of all the files and subfolders contained in the specified folder
Use:	intSize = *objFolder.*Size
Arguments:	*objFolder* Required. The name of a Folder Object previously instantiated.
Return:	The size of the specified folder
Remarks:	Size is in bytes
Example:	see below

```
Dim fso, f, myFolder
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFolder = $getAppPath()                                'Specify the app directory
Set f = fso.GetFolder (myFolder)
MsgBox "Size = " & f.Size & " bytes"                    'Display size of app folder
```

Property	**SubFolders**
Description:	Returns a **Folders** collection consisting of all folders contained in a specified folder,
Use:	objFC = *objFolder.*SubFolders
Arguments:	*objFolder* Required. The name of a Folder Object previously instantiated.
Return:	A folders collection of all subfolders in a specified folder.
Remarks:	Includes folders with hidden and system file attributes set.
Example:	see below

```
Dim fso, f, fc, s, myFolder
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFolder = $getAppPath()                                'Specify the app directory
Set f = fso.GetFolder (myFolder)
fc = f.Subfolders                                       'Returns collection of (sub)folders
s = ""
For each f1 in fc
    s = s & fc.name & vbCrLf
Next
MsgBox s
```

Property	**Type**
Description:	Returns information about the type of a folder.
Use:	strType = *objFolder.*Type
Arguments:	*objFolder* Required. The name of a Folder Object previously instantiated.
Return:	The type of folder.
Remarks:	If the object is a folder, "Folder" will be returned.
Example:	see below

```
Dim fso, f, myFolder
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFolder = $getAppPath()                                'Specify the app directory
Set f = fso.GetFolder (myFolder)
MsgBox "Type = " & f.Type                               'Displays "Folder"
```

File Object

The File Object allows the programmer refer to a specific file. Once the File object is instantiated, it can be referred to as an object from VBScript and its various Methods and Properties accessed.

The File Object is instantiated as follows:

```
Dim fso, f, myPath
Set fso = CreateObject("Scripting.FileSystemObject")   'Instantiate the FSO Object
myPath = $GetAppPath() & "notes.txt"                   'Define the path to the file of interest
Set f = fso.GetFile(myPath)                            'Instantiate the Drive Object
```

See the **GetFile** method under the FileSystemObject Object Model section for additional details on instantiation of the File Object.

The File object has both Methods and Properties available.

Method **Copy**

Description: Copies a specified file from one location to another.

Use: *objFile.*Copy (*destination,* [*overwrite*])

Arguments: *objFile*
: Required. The name of a File Object previously instantiated.

destination
: Required. Destination where the File is to be copied. Wildcard characters are not allowed.

overwrite
: Optional. Boolean value that is **True** (default) if existing files are to be overwritten, **False** if they are not.

Return: None

Remarks: The results of the **Copy** method on a **File** are identical to operations performed using **FileSystemObject.CopyFile** where the file referred to by *object* is passed as an argument. You should note, however, that the alternative method is capable of copying multiple files.

Example:

```
Dim fso, f, myFile
Set fso = CreateObject("Scripting.FileSystemObject")   'Instantiate the FSO object
myFile = $getAppPath() & "recipe1.xml"                 'Get the file object
Set f = fso.GetFile (myFile)
f.Copy ("c:\save\recipe1.xml")                         'Save the file
```

Method: **Delete**

Description: Deletes a specified file

Use: *objFile.*Delete (*force*)

Arguments: *objFile*
: Required. The name of a File Object previously instantiated.

force
: Optional. Boolean value that is **True** if files with the read-only attribute set are to be deleted; **False** if they are not (default).

Return: None

Remarks: An error occurs if the specified file does not exist. The results of the **Delete** method on a **File** are identical to operations performed using **FileSystemObject.DeleteFile**. The **Delete** method does not distinguish between files that have content and those that do not. The specified file is deleted regardless of whether or not it has content.

Example:

```
Dim fso, f, myFile
Set fso = CreateObject("Scripting.FileSystemObject")   'Instantiate the FSO object
myFile = $getAppPath() & "recipe1.xml"                 'Specify the HstTemp folder in app
directory
Set f = fso.GetFile (myFile)
f.Delete                                               'Delete it
```

Method:	**Move**
Description:	Moves a specified file from one location to another.
Use:	*objFile*.Move (*destination*)
Arguments:	*objFile* Required. The name of a File Object previously instantiated. *destination* Required. Destination where the file is to be moved. Wildcard characters are not allowed.
Return:	None
Remarks:	The results of the **Move** method on a **File** is identical to operations performed using **FileSystemObject.MoveFile**. You should note, however, that the alternative methods are capable of moving multiple files.

Example:

```
Dim fso, f, myFile
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFile = $getAppPath() & "recipe1.xml"                   'Specify the HstTemp folder in app
directory
Set f = fso.GetFile (myFile)
f.move("Recipe1Save.xml")                                'Moves the file
```

Method:	**OpenAsTextStream**
Description:	Opens a specified file name and returns a **TextStream** object that can be used to read from or write to, or append to a file
Use:	oTso = *oFile*.OpenAsTextStream([*iomode[,format]]*)
Arguments:	*objFile* Required. The name of a File Object previously instantiated. *iomode* Optional. Indicates the file input/output mode. Can be one of three constants:

Constant	Value	Description
ForReading	1	Open a file for reading only. You can't write to this file
ForWriting	2	Open a file for reading & writing
ForAppending	8	Open a file and write to the end of the file

format

Optional. One of three **Tristate** values used to indicate the format of the opened file. If omitted, the file is opened as ASCII.

Constant	Value	Description
TristateUseDefault	-2	Opens the file using the system default
TristateTrue	-1	Opens the file as Unicode
TrstateFalse	0	Opens the file as ASCII

Return:	A **TextStream** object
Remarks	The **OpenAsTextStream** method provides the same functionality as the **OpenTextFile** method of the **FileSystemObject**. In addition, the **OpenAsTextStream** method can be used to write to a file.

Example:

```
Const ForReading=1, Const ForWriting=2, ForAppending=8
Dim fso, f, tso
Set fso = CreateObject("Scripting.FileSystemObject")
Set f = fso.GetFile("c:\testfile.txt")              'Must be an existing file
Set tso = f.OpenAsTextStream(ForWriting, True)      'Unicode file
tso.Write "Hello world!"                            'Write a line of text to the file
tso.Close
```

Property: **Attributes**
Description: Sets or returns the attributes of files or folders.
Use: *objFile.*Attributes = newAttributes
intAttribute = *objFile*.Attributes
Arguments: *objFile*
Required. The name of a File Object previously instantiated.
newAttributes
Optional. If provided, newAttributes is the new value for the attributes of the specified *object*. The newattributes argument can have any of the following values or any logical combination of the following values:

Constant	Value	Description
Normal	0	Normal file. No Attributes are set.
ReadOnly	1	Read-only file. Attribute is read/write.
Hidden	2	Hidden file. Attribute is read/write.
System	4	System file. Attribute is read/write.
Volume	8	Disk drive volume label. Attribute is read-only
Directory	16	Folder or directory. Attribute is read-only.
Archive	32	File has changed since last backup. Attribute is read/write
Alias	1024	Link or shortcut. Attribute is read-only
Compressed	2048	Compressed file. Attribute is read-only.

Return: Can return an attribute of a file or folder
Remarks: Read/write or read-only, depending on the attribute. The newAttribute can have any valid combination of the above values.
Example:
```
Dim fso, f, attrVal, myFile
Set fso = CreateObject(“Scripting.FileSystemObject”)   ‘Instantiate the FSO object
myFile = $getAppPath() & “recipe1.xml”                 ‘Specify the app directory and file
Set f = fso.GetFile(myFile)
attrVal = f.Attributes
attrVal = attrVal And 1                                ‘See if a normal file
If attrVal = 0 Then
    MsgBox “Object is a normal file”
Else
    MsgBox “Object is not a normal file”
End If
```

Property: **DateCreated**
Description: Returns the date and time that the specified file was created.
Use: *objFile.*DateCreated
Arguments: *objFile*
Required. The name of a File Object previously instantiated.
Return: None
Remarks: Read-only.
Example:
```
Dim fso, f, myFile
Set fso = CreateObject(“Scripting.FileSystemObject”)   ‘Instantiate the FSO object
myFile = $getAppPath() & “recipe1.xml”                 ‘Specify the app directory & file
Set f = fso.GetFile (myFile)
MsgBox “File created on “ & f.DateCreated
```

Property: **DateLastAccessed**
Description: Returns the date and time that the specified file was last accessed
Use: *objFile.*DateLastAccessed
Arguments: *objFile*
    Required. The name of a File Object previously instantiated.
Return: None
Remarks: Read-only.
Example:

```
Dim fso, f, myFile
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFile = $getAppPath() & "recipe1.xml"                  'Specify the app directory & file
Set f = fso.GetFile (myFile)
MsgBox "File was last accessed on " & f.DateLastAccessed
```

Property: **DateLastModified**
Description: Returns the date and time that the specified file was last modified
Use: *objFile.*DateLastModified
Arguments: *objFile*
    Required. The name of a File Object previously instantiated.
Return: None
Remarks: Read-only.
Example:

```
Dim fso, f, myFile
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFile = $getAppPath() & "recipe1.xml"                  'Specify the app directory & file
Set f = fso.GetFile (myFile)
MsgBox "File was last modified on " & f.DateLastModified
```

Property: **Drive**
Description: Returns the drive letter of the drive on which the specified file resides
Use: *objFile.*Drive
Arguments: *objFile*
    Required. The name of a File Object previously instantiated.
Return: None
Remarks: Read-only.
Example:

```
Dim fso, f, myFile
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFile = $getAppPath() & "recipe1.xml"                  'Specify the app directory & file
Set f = fso.GetFile (myFile)
MsgBox "File is located on drive  " & f.Drive           'Installed on drive c:
```

Property: **Name**
Description: Sets or returns the name of a specified file
Use: *objFile.*Name = *newName*
strName = *objFile.*Name
Arguments: *objFile*
    Required. The name of a File Object previously instantiated.
newName
    Optional. If provided, *newName* is the new name of the specified file object
Return: The name of the specified file.
Remarks: Read/write.
Example:

```
Dim fso, f, myFile
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFile = $getAppPath() & "recipe1.xml"                  'Specify the app directory & file
Set f = fso.GetFile (myFile)
MsgBox "file name is " & f.Name                         'Returns the file name
```

Property:	**ParentFolder**	
Description:	Returns the folder object for the parent of the specified file	
Use:	objFolder = *objFile.*ParentFolder	
Arguments:	*objFile*	
	Required. The name of a File Object previously instantiated.	
Return:	The folder object for the parent folder of the specified file.	
Remarks:	Read-only	
Example:	Dim fso, f, pf, myFile	
	Set fso = CreateObject(“Scripting.FileSystemObject”)	‘Instantiate the FSO object
	myFile = $getAppPath() & “recipe1.xml”	‘Specify the app directory & file
	Set f = fso.GetFile (myFile)	
	Set pf = f.ParentFolder	‘Get the parent folder
	MsgBox “Parent Folder name = “ & pf.Name	

Property	**Path**	
Description:	Returns the path for a specified file	
Use:	strPath = *objFile.*Path	
Arguments:	*objFile*	
	Required. The name of a File Object previously instantiated.	
Return:	The path for a specified file	
Remarks:	None	
Example:	Dim fso, f, myFile	
	Set fso = CreateObject(“Scripting.FileSystemObject”)	‘Instantiate the FSO object
	myFile = $getAppPath() & “recipe1.xml”	‘Specify the app directory & file
	Set f = fso.GetFile (myFile)	
	MsgBox “Path = “ & UCase(f.Path)	‘Display path to app file

Property	**ShortName**	
Description:	Returns the short name used by programs that require the earlier 8.3 naming convention.	
Use:	strName = *objFile.*ShortName	
Arguments:	*objFile*	
	Required. The name of a File Object previously instantiated.	
Return:	The short name for the file object	
Remarks:	None	
Example:	Dim fso, f, myFile	
	Set fso = CreateObject(“Scripting.FileSystemObject”)	‘Instantiate the FSO object
	myFile = $getAppPath() & “recipe1.xml”	‘Specify the app directory & file
	Set f = fso.GetFile (myFile)	
	MsgBox “Short name = “ & f.ShortName	‘Display short name of app file

Property	**ShortPath**	
Description:	Returns the short path used by programs that require the earlier 8.3 naming convention.	
Use:	strPath = *objFile.*ShortPath	
Arguments:	*objFile*	
	Required. The name of a File Object previously instantiated.	
Return:	The short path for the file object	
Remarks:	None	
Example:	Dim fso, f, myFile	
	Set fso = CreateObject(“Scripting.FileSystemObject”)	‘Instantiate the FSO object
	myFile = $getAppPath() & “recipe1.xml”	‘Specify the app directory & file
	Set f = fso.GetFile (myFile)	
	MsgBox “Short name = “ & f.ShortPath	‘Display short path of app file

Property	**Size**
Description:	Returns the size of the specified file
Use:	intSize = *objFile.*Size
Arguments:	*objFile* Required. The name of a File Object previously instantiated.
Return:	The size of the specified file
Remarks:	Size is in bytes
Example:	

```
Dim fso, f, myFile
Set fso = CreateObject(“Scripting.FileSystemObject”)    ‘Instantiate the FSO object
myFile = $getAppPath() & “recipe1.xml”                  ‘Specify the app directory & file
Set f = fso.GetFile (myFile)
MsgBox “Size = “ & f.Size & “ bytes”                    ‘Display size of file
```

Property	**Type**
Description:	Returns information about the type of a file.
Use:	strType = *objFile.*Type
Arguments:	*objFile* Required. The name of a File Object previously instantiated.
Return:	The type of file.
Remarks:	E.g. for files ending in .TXT, "Text Document" is returned.
Example:	

```
Dim fso, f, myFile
Set fso = CreateObject(“Scripting.FileSystemObject”)    ‘Instantiate the FSO object
myFile = $getAppPath() & “recipe1.xml”                  ‘Specify the app directory & file
Set f = fso.GetFile (myFile)
MsgBox “Type = “ & f.Type                               ‘Dispays “XML Document”
```

TextStream Object

The TextStream Object allows the programmer to sequentially access a text file. Once the TextStream object is instantiated, it can be referred to as an object from VBScript and its various Methods and Properties accessed.

The TextStream object can be instantiated in three different ways. These are

- Through the **CreateTextFile** method of the FSO object
- Through the **OpenTextFile** method of the FSO object
- Through the **OpenAsTextStream** method of the File Object

There are subtle differences between these methods. The **CreateTextFile** is used to create a file and a TextStream object. This method can optionally overwrite an existing object. The **OpenTextFile** opens an existing file and returns a TextStream object, but can optionally create the filename if it does not exist. The **OpenAsTextStream** object opens an existing file and returns a TextStream object. This method gives an error if the text file does not exist, there is no option to create the file if it does not exist. Another difference is that the **CreateTextFile** method opens a TextStream object for reading and writing, while the **OpenTextFile** and **OpenAsTextStream** methods open a TextStream object for reading, writing or appending.

Examples of the various approaches to instantiating the TextStream object are:

Instantiating a TextStream object with the CreateTextFile Method

```
Dim fso, f, myfile
Set fso = CreateObject("Scripting.FileSystemObject")     'Instantiate the FSO object
myFile = $getAppPath() & "notes.txt"                     'Specify the app directory & file
Set f = fso.CreateTextFile(myFile, True, True)           'Open as UniCode TextStream object
```

Instantiating a TextStream object with the OpenTextFile Method

```
Constant forReading = 1, forWriting = 2, forAppending = 8
Dim fso, myfile, tso
Set fso = CreateObject("Scripting.FileSystemObject")     'Instantiate the FSO object
myFile = $getAppPath() & "notes.txt"                     'Specify the app directory & file
Set tso = fso.OpenTextFile(myFile, ForWriting, True, True) 'Open as UniCode TextStream object
```

Instantiating a TextStream object with the OpenAsTextStream Method

```
Constant forReading = 1, forWriting = 2, forAppending = 8
Dim fso, f, myfile, tso
Set fso = CreateObject("Scripting.FileSystemObject")     'Instantiate the FSO object
myFile = $getAppPath() & "notes.txt"                     'Specify the app directory & file
Set f = fso.GetFile(myFile)                              'Instantiate the file object
Set tso = f.OpenAsTextStream(forAppending, True)         'Open as UniCode TextStream object
```

See the **CreateTextFile** and **OpenTextFile** methods under the FileSystemObject Object Model section for additional details on instantiation of the TextStream Object. See the **OpenAsTextStream** method under the File Object section for additional details on instantiation of the TextStream Object

The TextStream object supports either ASCII or UniCode characters, according to the argument settings when calling the method used to instantiate the TextStream object.

Method: **Close**
Description: Closes an open TextStream file
Use: *objTso.*Close
Arguments: *objTso*
Required. The name of a TextStream Object previously instantiated.
Return: None
Remarks: The Close method closes the file, but still need to set the object variable to Nothing to release memory. (e.g. **"**Set objTso = Nothing"
Example:

```
Dim fso, f, myfile
Set fso = CreateObject("Scripting.FileSystemObject")   'Instantiate the FSO object
myFile = $getAppPath() & "notes.txt"                   'Specify the app directory & file
Set f = fso.CreateTextFile(myFile, True)
f.WriteLine ("this is a note")
f.Close                                                'Close the document
```

Method: **Read**
Description: Reads a specified number of characters from a **TextStream** file and returns the resulting string.
Use: strChars = *objTso.*Read(*numCharacters*)
Arguments: *objTso*
Required. The name of a TextStream Object previously instantiated.
numCharacters
Required. The number of characters you want to read from the file
Return: A specified number of characters from the file
Remarks: None
Example:

```
Const ForReading=1, Const ForWriting=2, ForAppending=8
Dim fso, f, myfile, s
Set fso = CreateObject("Scripting.FileSystemObject")   'Instantiate the FSO object
myFile = $getAppPath() & "notes.txt"                   'Specify the app directory & file
Set f = fso.OpenTextFile(myFile, ForReading)
s = f.Read(10)                                         'Read 10 characters
MsgBox "First 10 characters = " & s                    'Display
f.Close                                                'Close the document
```

Method: **ReadAll**
Description: Reads the entire **TextStream** file and returns the resulting string.
Use: strChars = *objTso.*ReadAll
Arguments: *objTso*
Required. The name of a TextStream Object previously instantiated.
Return: The entire TextStream file.
Remarks: VBScript does not have a limit on the resultant character string length other than the available memory.
Example:

```
Const ForReading=1, Const ForWriting=2, ForAppending=8
Dim fso, f, myfile, s
Set fso = CreateObject("Scripting.FileSystemObject")   'Instantiate the FSO object
myFile = $getAppPath() & "notes.txt"                   'Specify the app directory & file
Set f = fso.OpenTextFile(myFile, ForReading)
s = f.ReadAll                                          'Read entire file
MsgBox "File contents = " & s                          'Display it
f.Close
```

Method: **ReadLine**
Description: Reads an entire line (up to, but not including, the newline character) from a **TextStream** file and returns the resulting string.
Use: strChars = *objTso.*ReadLine
Arguments: *objTso*
Required. The name of a TextStream Object previously instantiated.
Return: An entire line from a TextStream file
Remarks: Does not include the newline character. Successive calls to the ReadLine method do not return any newline character(s). For display purposes, you must add a newline character
Example:

```
Const ForReading=1, Const ForWriting=2, ForAppending=8
Dim fso, f, myfile, s, linecount
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFile = $getAppPath() & "notes.txt"                    'Specify the app directory & file
Set f = fso.OpenTextFile(myFile, ForReading)
linecount = 0
s = ""
Do While f.AtEndOfStream <> True
    linecount = linecount +1
    s = s & "line " & linecount & "  " & f.ReadLine & vbCrLf        'Read a line at a time
Loop
MsgBox s                                                            'Display it
f.Close
```

Method: **Skip**
Description: Skips a specified number of characters when reading a **TextStream** file
Use: *objTso.Skip*(*numCharacters*)
Arguments: *objTso*
Required. The name of a TextStream Object previously instantiated.
numCharacters
Required. The number of characters you want to skip when reading a file
Return: None
Remarks: Skipped characters are discarded.
Example:

```
Const ForReading=1, Const ForWriting=2, ForAppending=8
Dim fso, f, myfile
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFile = $getAppPath() & "notes.txt"                    'Specify the app directory & file
Set f = fso.OpenTextFile(myFile, ForReading)
f.Skip(5)                                               'Skip 5 characters
MsgBox f.ReadLine                                       'Read the rest of the line
f.Close                                                 'Close the document
```

Method: **SkipLine**
Description: Skips the next line when reading from a **TextStream** file.
Use: *objTso.*SkipLine
Arguments: *objTso*
Required. The name of a TextStream Object previously instantiated.
Return: None
Remarks: The skipped line is discarded.
Example:

```
Const ForReading=1, Const ForWriting=2, ForAppending=8
Dim fso, f, myfile, s
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFile = $getAppPath() & "notes.txt"                    'Specify the app directory & file
Set f = fso.OpenTextFile(myFile, ForReading)
f.SkipLine                                              'Skip the first line
s=f.ReadLine
MsgBox s                                                'Display the second line
f.Close
```

Method: **Write**
Description: Writes a specified string to a **TextStream** file.
Use: *objTso.*Write(*string)*
Arguments: *objTso*
    Required. The name of a TextStream Object previously instantiated.
string
    Required. The text you want to write to the file.
Return: None
Remarks: Specified strings are written to the file with no intervening spaces or characters between each string. Use the **WriteLine** method to write a newline character or a string that ends with a newline character.
Example:

```
Const ForReading=1, Const ForWriting=2, ForAppending=8
Dim fso, f, myFile
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFile = $getAppPath() & "notes.txt"                    'Specify the app directory & file
Set f = fso.OpenTextFile(myFile, ForWriting, True)
f.Write "This is a new string of data"                  'Write a string
Set f = fso.OpenTextFile(myFile, ForReading)
MsgBox "File contents = " & f.ReadLine                  'Display line of data
f.Close
```

Method: **WriteBlankLines**
Description: Writes a specified number of newline characters to a **TextStream** file.
Use: *objTso.*WriteBlankLines(*numLines*)
Arguments: *objTso*
    Required. The name of a TextStream Object previously instantiated.
numLines
    Required. The number of newline characters you want to write to the file.
Return: None
Remarks: None
Example:

```
Const ForReading=1, Const ForWriting=2, ForAppending=8
Dim fso, f, myfile
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFile = $getAppPath() & "notes.txt"                    'Specify the app directory & file
Set f = fso.OpenTextFile(myFile, ForWriting, True)
f.WriteBlankLines(3)                                    'Write 3 blank lines
f.WriteLine "This is a new line of data"                'Write data on the 4th line
f.Close
```

Method: **WriteLine**
Description: Writes a specified string and newline character to a **TextStream** file.
Use: *objTso.*WriteLine([*string*])
Arguments: *objTso*
    Required. The name of a TextStream Object previously instantiated.
string
    Optional. The text you want to write to the file.
Return: None
Remarks: If you omit the *string*, a newline character is written to the file.
Example:

```
Const ForReading=1, Const ForWriting=2, ForAppending=8
Dim fso, f, myfile
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFile = $getAppPath() & "notes.txt"                    'Specify the app directory & file
Set f = fso.OpenTextFile(myFile, ForWriting, True)
f.WriteLine "This is a line of data"                    'Write a line of data
f.WriteLine                                             'Write a blank line
f.Close
```

Property: **AtEndOfLine**
Description: Indicates whether the file pointer is positioned immediately before the end-of-line marker in a **TextStream** file.
Use: *objTso.*AtEndOfLine
Arguments: *objTso*
    Required. The name of a TextStream Object previously instantiated.
Return: Returns **True** if the file pointer is positioned immediately before the end-of-line marker in a **TextStream** file; **False** if it is not.
Remarks: The **AtEndOfLine** property applies only to **TextStream** files that are open for reading; otherwise, an error occurs.
Example:

```
Const ForReading=1, Const ForWriting=2, ForAppending=8
Dim fso, f, myfile, s
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFile = $getAppPath() & "notes.txt"                    'Specify the app directory & file
Set f = fso.OpenTextFile(myFile, ForReading, False)
s =""
Do While f.AtEndOfLine <> True
    s=f.read(1)                                         'Read one character at a time
Loop
MsgBox "A line of text = " & s
f.Close
```

Property: **AtEndOfStream**
Description: Indicates whether the file pointer is positioned at the end of a **TextStream** file.
Use: *objTso.*AtEndOfStream
Arguments: *objTso*
    Required. The name of a TextStream Object previously instantiated.
Return: Returns **True** if the file pointer is positioned at the end of a **TextStream** file; **False** if it is not.
Remarks: The **AtEndOfStream** property applies only to **TextStream** files that are open for reading; otherwise, an error occurs.
Example:

```
Const ForReading=1, Const ForWriting=2, ForAppending=8
Dim fso, f, myfile, s
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFile = $getAppPath() & "notes.txt"                    'Specify the app directory & file
Set f = fso.OpenTextFile(myFile, ForReading, False)
s =""
Do While f.AtEndOfLine <> True
    s = s & f.ReadLine                                  'Read file one line at a time
Loop
MsgBox s                                                'Display text
f.Close
```

Property: **Column**
Description: Returns the column number of the current character position in a **TextStream** file.
Use: intColumnPos = *objTso.*Column
Arguments: *objTso*
    Required. The name of a TextStream Object previously instantiated.
Return: An integer column number
Remarks: Read-only. After a newline character has been written, but before any other character is written, **Column** is equal to 1.
Example:

```
Const ForReading=1, Const ForWriting=2, ForAppending=8
Dim fso, f, myfile, s, colNum
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFile = $getAppPath() & "notes.txt"                    'Specify the app directory & file
Set f = fso.OpenTextFile(myFile, ForReading, False)
s = f.ReadLine                                          'Read a line
colNum = f.Column                                       'Get the column position
f.Close
```

Property:	**Line**
Description:	Returns the current line number in a **TextStream** file.
Use:	intLineNum = *objTso.*Line
Arguments:	*objTso* Required. The name of a TextStream Object previously instantiated.
Return:	An integer line number
Remarks:	Read-only. After a file is initially opened and before anything is written, **Line** is equal to 1.

Example:

```
Const ForReading=1, Const ForWriting=2, ForAppending=8
Dim fso, f, myfile, s, lineNum
Set fso = CreateObject("Scripting.FileSystemObject")    'Instantiate the FSO object
myFile = $getAppPath() & "notes.txt"                    'Specify the app directory & file
Set f = fso.OpenTextFile(myFile, ForReading, False)
s = f.ReadAll                                           'Read the entire file
lineNum = f.Line                                        'Get the last line number
f.Close
```