Feature Information

JScript Features - ECMA
JScript Features - Non-ECMA
Microsoft Scripting Run-Time Features
$1...$9 Properties
abs Method
acos Method
ActiveXObject Object
Addition Operator (+)
anchor Method
arguments Property
Array Object
asin Method
Assignment Operator (=)
atan Method
atan2 Method
atEnd Method
big Method
Bitwise AND Operator (&)
Bitwise Left Shift Operator (<<)
Bitwise NOT Operator (~)
Bitwise OR Operator (|)
Bitwise Right Shift Operator (>>)
Bitwise XOR Operator (^)
blink Method
bold Method
Boolean Object
break Statement
caller Property
catch Statement
@cc_on Statement
ceil Method
charAt Method
charCodeAt Method
Comma Operator (,)
// (Single-line Comment Statement)
/*..*/ (Multiline Comment Statement)
Comparison Operators
compile Method
Compound Assignment Operators
concat Method (Array)
concat Method (String)
Conditional Compilation
Conditional Compilation Variables
Conditional (trinary) Operator (?:)
constructor Property
continue Statement
cos Method
Data Type Conversion
Date Object
Decrement Operator (--)
delete Operator
description Property
Dictionary Object
dimensions Method
Division Operator (/)
do...while Statement
E Property
Enumerator Object
Equality Operator (==)
Error Object
escape Method
eval Method
exec Method
exp Method
FileSystemObject Object
fixed Method
floor Method
fontcolor Method
fontsize Method
for Statement
for...in Statement
fromCharCode Method
Function Object
function Statement
getDate Method
g.getDay Method
g.getFullYear Method
g.getfullYear Method
g.getHours Method
g.getItem Method
g.getMilliseconds Method
g.getMinutes Method
g.getMonth Method
GetObject Function
g.getSeconds Method
g getTime Method
g.getTimezoneOffset Method
g.getUTCDate Method
g.getUTCDay Method
g.getUTCFullYear Method
g.getUTCHours Method
g.getUTCMilliseconds Method
g.getUTCMinutes Method
g.getUTCMonth Method
g.getUTCSeconds Method
g.getVarDate Method
g.getYear Method
Global Object
Greater than Operator (>)
Greater than or equal to Operator (>=)
Identity Operator (===)
@if Statement
if...else Statement
Increment Operator (++)
index Property
indexOf Method
Inequality Operator (!=)
Infinity Property
input Property
instanceof Operator
isFinite Method
isNaN Method
italics Method
item Method
join Method
Labeled Statement
lastIndex Property
lastIndexOf Method
lbound Method
length Property (Array)
length Property (Function)
length Property (String)
Less than Operator (<)
Less than or equal to Operator (<=)
link Method
LN2 Property
LN10 Property
log Method
LOG2E Property
LOG10E Property
Logical AND Operator (&&)
Logical NOT Operator (!)
Logical OR Operator (||)
match Method
Math Object
max Method
MAX_VALUE Property
min Method
MIN_VALUE Property
Modulus Operator (%)
moveFirst Method
moveNext Method
Multiplication Operator (*)
NaN Property (Global)
NaN Property (Number)
NEGATIVE_INFINITY Property
new Operator
Nonidentity Operator (!==)
Number Object
number Property
Object Object
Operator Precedence
parse Method
parseFloat Method
parseInt Method
PI Property
POSITIVE_INFINITY Property
pow Method
prototype Property
random Method
RegExp Object
Regular Expression Object
Regular Expression Syntax
replace Method
return Statement
reverse Method
round Method
ScriptEngine Function
ScriptEngineBuildVersion Function
ScriptEngineMajorVersion Function
ScriptEngineMinorVersion Function
search Method
@set Statement
setDate Method
setFullYear Method
setHours Method
setMilliseconds Method
setMinutes Method
setMonth Method
setSeconds Method
setTime Method
setUTCDate Method
setUTCFullYear Method
setUTCHours Method
setUTCMilliseconds Method
setUTCMinutes Method
setUTCMonth Method
setUTCSeconds Method
setYear Method
sin Method
slice Method (Array)
slice Method (String)
small Method
sort Method
source Property
split Method
sqrt Method
SQRT1_2 Property
SQRT2 Property
strike Method
String Object
sub Method
substr Method
substring Method
Subtraction Operator (-)
sup Method
switch Statement
tan Method
test Method
this Statement
throw Statement
toArray Method
toGMTString Method
toLocaleString Method
toLowerCase Method
toString Method
toUpperCase Method
toUTCString Method
try Statement
typeof Operator
ubound Method
Unary Negation Operator (-)
unescape Method
Unsigned Right Shift Operator (>>>)
UTC Method
valueOf Method
var Statement
VBAArray Object
void Operator
while Statement
with Statement
JScript Errors

Run-Time Errors
Syntax Errors
Functions

GetObject Function
ScriptEngine Function
ScriptEngineBuildVersion Function
ScriptEngineMajorVersion Function
ScriptEngineMinorVersion Function
abs Method
acos Method
anchor Method
asin Method
atan Method
atan2 Method
AtEnd Method
big Method
blink Method
bold Method
ceil Method
charAt Method
charCodeAt Method
compile Method
concat Method (Array)
concat Method (String)
cos Method
dimensions Method
escape Method
eval Method
exec Method
exp Method
fixed Method
floor Method
fontcolor Method
fontsize Method
fromCharCode Method
getTime Method
getDay Method
getFullYear Method
getHours Method
getItem Method
getMilliseconds Method
getMinutes Method
getMonth Method
getSeconds Method
getTime Method
getTimezoneOffset Method
getUTCDate Method
getUTCDate Method
getUTCFullYear Method
getUTCFullYear Method
getUTCDate Method
getUTCDate Method
getUTCDate Method
getUTCDate Method
getVarDate Method
getYear Method
indexOf Method
isFinite Method
isNaN Method
italics Method
item Method
join Method
lastIndexOf Method
lbound Method
link Method
log Method
match Method
max Method
min Method
moveFirst Method
moveNext Method
parse Method
parseFloat Method
parseInt Method
pow Method
random Method
replace Method
reverse Method
round Method
search Method
setDate Method
setFullYear Method
setHours Method
setMilliseconds Method
setMinutes Method
setMonth Method
setSeconds Method
setTime Method
setUTCDate Method
setUTCFullYear Method
setUTCHours Method
setUTCMilliseconds Method
setUTCMinutes Method
setUTCMilliseconds Method
setUTCMonth Method
setUTCSeconds Method
setYear Method
sin Method
slice Method (Array)
slice Method (String)
small Method
sort Method
split Method
sqrt Method
strike Method
sub Method
substr Method
substring Method
sup Method
tan Method
test Method
toArray Method
toGMTString Method
toLocaleString Method
toLowerCase Method
toString Method
toUpperCase Method
toUTCString Method
ubound Method
unescape Method
UTC Method
valueOf Method
ActiveXObject Object
Array Object
Boolean Object
Date Object
Dictionary Object
Enumerator Object
Error Object
FileSystemObject Object
Function Object
Global Object
Math Object
Number Object
Object Object
RegExp Object
RegularExpression Object
String Object
VBAArray Object
Addition Operator (+)
Assignment Operator (=)
Bitwise AND Operator (&)
Bitwise Left Shift Operator (<<)
Bitwise NOT Operator (~)
Bitwise OR Operator (|)
Bitwise Right Shift Operator (>>)
Bitwise XOR Operator (^)
Comma Operator (,)
Comparison Operators
Compound Assignment Operators
Conditional (trinary) Operator (?:)
Decrement Operator (--) 
delete Operator
Division Operator (/)
Equality Operator (==)
Greater than Operator (>)
Greater than or equal to Operator (>=)
Identity Operator (===)
Increment Operator (++)
Inequality Operator (!=)
instanceof Operator
Less than Operator(<)
Less than or equal to Operator (<=)
Logical AND Operator (&&)
Logical NOT Operator (!)
Logical OR Operator (||)
Modulus Operator (%)
Multiplication Operator (*)
new Operator
Nonidentity Operator (!==)
Operator Precedence
Subtraction Operator (-)
typeof Operator
Unary Negation Operator (-)
Unsigned Right Shift Operator (>>>)
void Operator
Properties

$1...$9 Properties
arguments Property
caller Property
constructor Property
description Property
E Property
index Property
Infinity Property
input Property
lastIndex Property (RegExp)
length Property (Array)
length Property (Function)
length Property (String)
LN2 Property
LN10 Property
LOG2E Property
LOG10E Property
MAX_VALUE Property
MIN_VALUE Property
Nan Property (Global)
NaN Property (Number)
NEGATIVE_INFINITY Property
number Property
PI Property
POSITIVE_INFINITY Property
prototype Property
source Property
SQRT1_2 Property
SQRT2 Property
Statements

break Statement  
catch Statement  
@cc_on Statement  
Comment Statements  
continue Statement  
do...while Statement  
for Statement  
for...in Statement  
function Statement  
@if Statement  
if...else Statement  
Labeled Statement  
return Statement  
@set Statement  
switch Statement  
this Statement  
throw Statement  
try Statement  
var Statement  
while Statement  
with Statement
JScript is the Microsoft implementation of the ECMA 262 language specification. It is a full implementation, plus some enhancements that take advantage of capabilities of Microsoft Internet Explorer. This tutorial is intended to help you get started with JScript.

**Easy to Use, Easy to Learn**

JScript is an interpreted, object-based scripting language. Although it has fewer capabilities than full-fledged object-oriented languages like C++ and Java, JScript is more than sufficiently powerful for its intended purposes.

JScript is not a cut-down version of any other language (it is only distantly and indirectly related to Java, for example), and it is not a simplification of anything. It is, however, limited. You cannot write standalone applications in it, for example, and it has little capability for reading or writing files. Moreover, JScript scripts can run only in the presence of an interpreter, either in a Web server or a Web browser.

JScript is a loosely typed language. That means you do not have to declare the data types of variables explicitly. In fact, you cannot explicitly declare data types in JScript. Moreover, in many cases JScript performs conversions automatically when they are needed. For instance, if you try to add a number to an item that consists of text (a string), the number is converted to text.

The rest of this tutorial is an overview of JScript features. For full details of the language implementation, consult the [language reference](#).
**Note** The code in many of this tutorial's examples is somewhat more explicit and less dense than code you'll find in actual Web pages. Most of it is also fairly simple. The intent here is to clarify the concepts, not to express optimal coding conciseness and style. There is, in any case, no shame in writing code that you can read and easily understand, six months after you write it.
Like many other programming languages, Microsoft JScript is written in text format, and is organized into statements, blocks consisting of related sets of statements, and comments. Within a statement you can use variables, immediate data such as strings and numbers, and expressions.

**Statements**

A JScript code statement consists of one or more items and symbols on a line. A new line begins a new statement, but it is a good idea to terminate your statements explicitly. You can do this with the semicolon (;), which is the JScript termination character.

```javascript
aBird = "Robin";
var today = new Date();
```

A group of JScript statements that is surrounded by braces ({}) is called a block. Blocks of statements are used, for example, in functions and conditionals.

In the following example, the first statement begins the definition of a function, which consists of a block of five statements. The last three statements, which are not surrounded by braces, are not a block and are not part of the function definition.

```javascript
function convert(inches) {
    feet = inches / 12; // These five statements are in a
```
miles = feet / 5280;
nauticalMiles = feet / 6080;
cm = inches * 2.54;
meters = inches / 39.37;
}
km = meters / 1000;  // These three statements are not
kradius = km;
mradius = miles;

Comments

A single-line JScript comment begins with a pair of forward
slashes (//). A multiline comment begins with a forward slash and
asterisk in combination (/*), and ends with the reverse (*/).

aGoodIdea = "Comment your code thoroughly.";  // T

/*
This is a multiline comment that explains the precedin

The statement assigns a value to the aGoodIdea variab
is contained between the quote marks, is called a litera
and directly contains information; it does not refer to t
(The quote marks are not part of the literal.)
*/

// This is another multiline comment, written as a seri
// After the statement is executed, you can refer to the
// variable by using its name, as in the next statement,  
// appended to the aGoodIdea variable by concatenation

var extendedIdea = aGoodIdea + " You never know when"

Assignments and Equality

The equal sign (=) is used in JScript to indicate the action of assigning a value. That is, a JScript code statement could say

    anInteger = 3;

It means "Assign the value 3 to the variable anInteger," or "anInteger takes the value 3." When you want to compare two values to find out whether they are equal, use a pair of equal signs (==). This is discussed in detail in Controlling Program Flow.

Expressions

A JScript expression is something that a person can read as a Boolean or numeric expression. Expressions contain symbol characters like "+" rather than words like "added to". Any valid combination of values, variables, operators, and expressions constitutes an expression.

    var anExpression = "3 * (4 / 5)";
    var aSecondExpression = "Math.PI * radius * 2";
    var aThirdExpression = aSecondExpression + "%" + a
    var aFourthExpression = "(" + aSecondExpression + "]" + aSecondExpression + "]"
Variables are used in Microsoft JScript to store values in your scripts. They are a way to retrieve and manipulate values using names. When used effectively then can help in understanding what a script does.

**Declaring Variables**

Although not required, it is considered good practice to declare variables before using them. You do this using the `var` statement. The only time you must use the `var` statement is when declaring variables that are local to a function. Local variables are those that are only within the function. At all other times, using the `var` statement to declare variables before their use is a recommended practice.

The following code examples are of variable declaration:

```javascript
var mim = "A man, a plan, a canal, Panama!"; // The value stored in mim is of string type.
// The sentence in quotes, the value of which is assigned to mim, is a string literal.
var ror = 3; // The value stored in ror has numeric type.
var nen = true; // The value stored in nen has Boolean type.
var fif = 2.718281828; // The value stored in fif has numeric type.
```

**Naming Variables**
JScript is a case-sensitive language, so naming a variable `myCounter` is different from naming it `MYCounter`. In addition, variable names, which can be of any length, must follow certain rules:

- The first character must be a letter (either uppercase or lowercase) or an underscore (_), or a dollar sign ($).
- Subsequent characters can be letters, numbers, underscores, or dollar signs.
- The variable name can't be a reserved word.

Some examples of valid variable names:

- `_pagecount`
- `Part9`
- `Number_Items`

Some invalid variable names:

- `99Balloons` // Starts with a number.
- `Smith&Wesson;` // Ampersand (&) is not a valid character for variable names.

In instances in which you want to declare a variable and initialize it, but without giving it any particular value, you may assign it a special value, `null`.

```javascript
var zaz = null;
var notalot = 3 * zaz; // At this point, notalot beco
```

If you declare a variable without assigning any value to it, it
exists but is **undefined**.

```javascript
var godot;
var waitingFor = 1 * godot; // Places the value NaN in waitingFor as godot is undefined.
```

You can declare a variable implicitly (without using `var`) by assigning a value to it. You cannot, however, use a variable that has never been declared at all. To do so generates an error at runtime.

```javascript
lel = ""; // The variable lel is declared implicitly.
```

```javascript
var aMess = vyv + zez; // Generates an error because vyv and zez don't exist.
```

**Coercion**

As JScript is a loosely-typed language, variables in JScript technically have no fixed type. Instead, they have a type equivalent to the type of the value they contain. It is possible, under some circumstances, to force the automatic conversion (or coercion) of a variable or a piece of data into a different type. Numbers can easily be included in strings, but strings cannot be included directly in numbers, so explicit conversion functions, `parseInt()` and `parseFloat()`, are provided.

```javascript
var theFrom = 1;
var theTo = 10;
var doWhat = "Count from ";
doWhat += theFrom + " to " + theTo + ".";
```

After this code is executed, the `doWhat` variable contains "Count
from 1 to 10." The number data have been coerced into string form.

```javascript
var nowWhat = 0;
nowWhat += 1 + "10";  // In this case, because "10" is
// the "+=" operator concatenates.
```

After this code is executed, the `nowWhat` variable contains "0110". The following steps are followed to arrive at this result:

1. Look at the types of 1 and "10". The "10" is a string, the 1 is a number, so the number is coerced into a string.
2. As the values on either side of the + operator are both strings, do a string concatenation. This results in "110"
3. Look at the types of the values on either side of the +=.
   `nowWhat` contains a number, and "110" is a string, so convert the number to a string.
4. As there are now strings on either side of the += operator, do a string concatenation. This results in "0110".
5. Store this result in `nowWhat`.

```javascript
var nowThen = 0;
nowThen += 1 + parseInt("10");  // In this case, "+
```

After this code is executed, the `nowThen` variable contains the integer 11.
What Are the JScript Data Types?

Microsoft JScript has six types of data. The main types are numbers, strings, objects, and Booleans. The other two are null and undefined.

String Data Type

Strings are delineated by single or double quotation marks. (Use single quotes to type strings that contain quotation marks.) A string is also an object in JScript, but it is a special case, with special properties. The following are examples of strings:

"The cow jumped over the moon."
""Avast, ye lubbers!" roared the technician.'
"42"

A string can contain zero or more unicode characters. When it contains zero, it is called a zero-length string ("'").

Number Data Type

JScript supports both integer and floating-point numbers. Integers can be positive, 0, or negative; a floating-point number
can contain either a decimal point, an "e" (uppercase or lowercase), which is used to represent "ten to the power of" in scientific notation, or both. These numbers follow the IEEE 754 standard for numerical representation. Last, there are certain number values that are special:

- **NaN**, or not a Number
- Positive Infinity
- Negative Infinity
- Positive 0
- Negative 0

Integers can be represented in base 10 (decimal), base 8 (octal), and base 16 (hexadecimal).

Octal integers are specified by a leading "0", and can contain digits 0 through 7. If a number has a leading "0" but contains the digits "8" and/or "9", it is a decimal number. A number that would otherwise be an octal number but contains the letter "e" (or "E") generates an error.

Hexadecimal ("hex") integers are specified by a leading "0x" (the "X" can be uppercase or lowercase) and can contain digits 0 through 9 and letters A through F (either uppercase or lowercase). The letter "e" is a permissible digit in hexadecimal notation and does not signify an exponential number. The letters A through F are used to represent, as single digits, the numbers that are 10 through 15 in base 10. That is, 0xF is equivalent to 15, and 0x10 is equivalent to 16.

Octal and hexadecimal numbers can be negative, but cannot be fractional. A number that begins with a single "0" and contains a decimal point is a decimal floating-point number; if a number that begins with "0x" or "00" contains a decimal point, anything to the right of the decimal point is ignored.
Some example numbers:

.0001, 0.0001, 1e-4, 1.0e-4 // Four floating-point numbers.
3.45e2 // A floating-point number, equivalent to 345.
42 // An integer number.
0377 // An octal integer, equivalent to 255.
00.0001 // As octal numbers cannot have decimal parts, this is equivalent to 0.
0378 // An integer, equivalent to 378.
0Xff // A hexadecimal integer, equivalent to 255.
0x37CF // A hexadecimal integer, equivalent to 14287.
0x3e7 // A hexadecimal integer, equivalent to 999.
0x3.45e2 // As hexadecimal numbers cannot have decimal parts, this is equivalent to 3.

Booleans

The possible Boolean values are **true** and **false**. These are special values, and are not usable as 1 and 0.

---

**Note** In a comparison, any expression that evaluates to 0 is taken to be false, and any statement that evaluates to a number other than 0 is taken to be true. Thus the following expression evaluates to true:

```
(false == 0)
```

---

For more information on comparisons, see [Controlling Program Flow](#).

Undefined Data Type
A value that is undefined is simply a value given to a variable after it has been created, but before a value has been assigned to it.

**Null Data Type**

A **null** value is one that has no value and means nothing.
JScript has a full range of operators, including arithmetic, logical, bitwise, and assignment operators. There are also a few miscellaneous operators.

<table>
<thead>
<tr>
<th>Computational</th>
<th>Logical</th>
<th>Bitwise</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Symbol</td>
<td>Description</td>
<td>Symbol</td>
</tr>
<tr>
<td>Unary negation</td>
<td>-</td>
<td>Logical NOT</td>
<td>!</td>
</tr>
<tr>
<td>Increment</td>
<td>++</td>
<td>Less than</td>
<td>&lt;</td>
</tr>
<tr>
<td>Decrement</td>
<td>--</td>
<td>Greater than</td>
<td>&gt;</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
<td>Less than or equal to</td>
<td>&lt;=</td>
</tr>
<tr>
<td>Division</td>
<td>/</td>
<td>Greater than or equal to</td>
<td>&gt;=</td>
</tr>
<tr>
<td>Modulus arithmetic</td>
<td>%</td>
<td>Equality</td>
<td>==</td>
</tr>
<tr>
<td>Addition</td>
<td>+</td>
<td>Inequality</td>
<td>!=</td>
</tr>
<tr>
<td>Subtraction</td>
<td>-</td>
<td>Logical AND</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Conditional (trinary)</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Identity</td>
<td>===</td>
</tr>
</tbody>
</table>

**Table: JScript Operators**
Operator Precedence

Operators in JScript are evaluated in a particular order. This order is known as the operator precedence. The following table lists the operators in highest to lowest precedence order. Operators in the same row are evaluated in left to right order.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>. [] ()</td>
<td>Field access, array indexing, and function calls</td>
</tr>
<tr>
<td>++ -- - ~ ! typeof new void delete</td>
<td>Unary operators, return data type, object creation, undefined values</td>
</tr>
<tr>
<td>* / %</td>
<td>Multiplication, division, modulo division</td>
</tr>
<tr>
<td>+ - +</td>
<td>Addition, subtraction, string concatenation</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt; &gt;&gt;&gt;</td>
<td>Bit shifting</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>Less than, less than or equal to, greater than, greater than or equal to</td>
</tr>
<tr>
<td>== != === !==</td>
<td>Equality, inequality, identity, nonidentity</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>Conditional</td>
</tr>
<tr>
<td>= OP=</td>
<td>Assignment, assignment with operation</td>
</tr>
<tr>
<td>,</td>
<td>Multiple evaluation</td>
</tr>
</tbody>
</table>

Parentheses are used to alter the order of evaluation. The expression within parentheses is fully evaluated before its value is used in the remainder of the statement.
An operator with higher precedence is evaluated before one with lower precedence. For example:

\[ z = 78 \times (96 + 3 + 45) \]

There are five operators in this expression: =, *, (), +, and +. According to precedence, they are evaluated in the following order: (), *, +, +, =.

1. Evaluation of the expression within the parentheses is first: There are two addition operators, and they have the same precedence: 96 and 3 are added together and 45 is added to that total, resulting in a value of 144.

2. Multiplication is next: 78 and 144 are multiplied, resulting in a value of 11232.

3. Assignment is last: 11232 is assigned into z.
Why Control the Flow of Execution?

Fairly often, you need a script to do different things under different conditions. For example, you might write a script that checks the time every hour, and changes some parameter appropriately during the course of the day. You might write a script that can accept some sort of input, and act accordingly. Or you might write a script that repeats a specified action.

There are several kinds of conditions that you can test. All conditional tests in Microsoft JScript are Boolean, so the result of any test is either true or false. You can freely test values that are of Boolean, numeric, or string type.

JScript provides control structures for a range of possibilities. The simplest control structures are the conditional statements.

Using Conditional Statements

JScript supports if and if...else conditional statements. In if statements a condition is tested, and if the condition meets the test, some JScript code you've written is executed. (In the if...else statement, different code is executed if the condition fails the test.) The simplest form of an if statement can be written entirely on one line, but multiline if and if...else statements are much more common.

The following examples demonstrate syntaxes you can use with if and
if...else statements. The first example shows the simplest kind of Boolean test. If (and only if) the item between the parentheses evaluates to true, the statement or block of statements after the if is executed.

    // The smash() function is defined elsewhere in the code.
    if (newShip)
        smash(champagneBottle, bow);  // Boolean test of whether newShip is true.

    // In this example, the test fails unless both conditions
    if (rind.color == "deep yellow " && rind.texture == ""
    {  
        theResponse = ("Is it a Crenshaw melon? <br> ")
    }  

    // In this example, the test succeeds if either condition
    var theReaction = "";
    if ((lbsWeight > 15) || (lbsWeight > 45))
    {  
        theReaction = ("Oh, what a cute kitty! <br>");
    }
    else
        theReaction = ("That's one huge cat you've got there.

Conditional Operator

JScript also supports an implicit conditional form. It uses a question mark after the condition to be tested (rather than the word if before the condition), and specifies two alternatives, one to be used if the condition is met and one if it is not. The
alternatives are separated by a colon.

```javascript
var hours = "";

// Code specifying that hours contains either the contents of
// theHour, or theHour - 12.

hours += (theHour >= 12) ? " PM" : " AM";
```

---

**Tip**  If you have several conditions to be tested together, and you know that one is more likely to pass or fail than any of the others, depending on whether the tests are connected with OR (||) or AND (&&), you can speed execution of your script by putting that condition first in the conditional statement. For example, if three conditions must all be true (using && operators) and the second test fails, the third condition is not tested.

Similarly, if only one of several conditions must be true (using || operators), testing stops as soon as any one condition passes the test. This is particularly effective if the conditions to be tested involve execution of function calls or other code.

An example of the side effect of short-circuiting is that runsecond will not be executed in the following example if runfirst() returns 0 or `false`.

```javascript
if ((runfirst() == 0) || (runsecond() == 0))
  // some code
```

---

**Using Repetition, or Loops**

There are several ways to execute a statement or block of statements repeatedly. In general, repetitive execution is called *looping*. It is typically controlled by a test of some variable, the value of which is changed each time the loop is executed.
Microsoft JScript supports many types of loops: **for** loops, **for...in** loops, **while** loops, **do...while** loops, and **switch** loops.

### Using for Loops

The **for** statement specifies a counter variable, a test condition, and an action that updates the counter. Just before each time the loop is executed (this is called one pass or one iteration of the loop), the condition is tested. After the loop is executed, the counter variable is updated before the next iteration begins.

If the condition for looping is never met, the loop is never executed at all. If the test condition is always met, an infinite loop results. While the former may be desirable in certain cases, the latter rarely is, so take care when writing your loop conditions.

```javascript
/*
The update expression ("icount++" in the following example) is executed at the end of the loop, after the block of statements that forms the body of the loop is executed, and before the condition is tested.
*/

var howFar = 11; // Sets a limit of 11 on the loop.
var sum = new Array(howFar); // Creates an array called sum
var theSum = 0;
sum[0] = 0;

for(var icount = 1; icount < howFar; icount++) {
    theSum += icount;
}
sum[icount] = theSum;
}

var newSum = 0;
for(var icount = 1; icount > howFar; icount++) {
newSum += icount;
}

var sum = 0;
for(var icount = 1; icount > 0; icount++) {
  // This is an infinite loop.
  sum += icount;
}

Using for...in Loops

JScript provides a special kind of loop for stepping through all the properties of an object. The loop counter in a for...in loop steps through all indexes in the array. It is a string, not a number.

for (j in tagliatelleVerde) // tagliatelleVerde is an object
{
  // Various JScript code statements.
}

Using while Loops

The while loop is very similar to a for loop. The difference is that a while loop does not have a built-in counter variable or update expression. If you already have some changing condition
that is reflected in the value assigned to a variable, and you want to use it to control repetitive execution of a statement or block of statements, use a **while** loop.

```javascript
var theMoments = "";
var theCount = 42;    // Initialize the counter variable.
while (theCount >= 1) {
    if (theCount > 1) {
        theMoments = "Only " + theCount + " moments!";
    }
    else {
        theMoments = "Only one moment left!";
    }
    theCount--;      // Update the counter variable.
}
theMoments = "BLASTOFF!";
```

**Note** Because **while** loops do not have explicit built-in counter variables, they are even more vulnerable to infinite looping than the other types. Moreover, partly because it is not necessarily easy to discover where or when the loop condition is updated, it is only too easy to write a **while** loop in which the condition, in fact, never does get updated. You should be extremely careful when you design **while** loops.

---

**Using break and continue Statements**

Microsoft JScript provides a statement to stop the execution of a
loop. The **break** statement can be used to stop execution if some (presumably special) condition is met. The **continue** statement can be used to jump immediately to the next iteration, skipping the rest of the code block but updating the counter variable as usual if the loop is a **for** or **for...in** loop.

```javascript
var theComment = ""; 
var theRemainder = 0; 
var theEscape = 3; 
var checkMe = 27; 
for (kcount = 1; kcount <= 10; kcount++)
{
    theRemainder = checkMe % kcount; 
    if (theRemainder == theEscape)
    {
        break; // Exits from the loop at the first encounter
    }
    theComment = checkMe + " divided by " + kcount + " 
}
```

```javascript
for (kcount = 1; kcount <= 10; kcount++)
{
    theRemainder = checkMe % kcount; 
    if (theRemainder != theEscape)
    {
        continue; // Selects only those remainders that equal the escape
    }
}
```
// JScript code that uses the selected remainders.
}

var theMoments = "";
var theCount = 42; // The counter is initialized.
while (theCount >= 1) {
   // if (theCount < 10) { // Warning!
   // This use of continue creates an infinite loop!
   // continue;
   // }
   if (theCount > 1) {
      theMoments = "Only " + theCount + " moments!
   }
   else {
      theMoments = "Only one moment left!";
   }
   theCount--; // The counter is updated.
}
theCount = "BLASTOFF!";
What Is a Function?

Microsoft JScript functions perform actions. They can also return results. Sometimes these are the results of calculations or comparisons.

Functions combine several operations under one name. This lets you streamline your code. You can write out a set of statements, name it, and then execute the entire set any time you want to, just by calling it and passing to it any information it needs.

You pass information to a function by enclosing the information in parentheses after the name of the function. Pieces of information that are being passed to a function are called arguments or parameters. Some functions don't take any arguments at all; some functions take one argument; some take several. There are even functions for which the number of arguments depends on how you are using the function.

JScript supports two kinds of functions: those that are built into the language, and those you create yourself.

Special Built-in Functions

The JScript language includes several built-in functions. Some of them let you handle expressions and special characters, and convert strings to numeric values.

For example, escape() and unescape() are used to convert characters that have special meanings in HTML code, characters that you cannot just put directly into text. For example, the angle brackets, "<" and ">", delineate
The `escape` function takes as its argument any of these special characters, and returns the escape code for the character. Each escape code consists of a percent sign (%) followed by a two-digit number. The `unescape` function is the exact inverse. It takes as its argument a string consisting of a percent sign and a two-digit number, and returns a character.

Another useful built-in function is `eval()`, which evaluates any valid mathematical expression that is presented in string form. The `eval()` function takes one argument, the expression to be evaluated.

```javascript
var anExpression = "6 * 9 % 7";
var total = eval(anExpression);  // Assigns the value 5 to the variable total.
var yetAnotherExpression = "6 * (9 % 7)";
total = eval(yetAnotherExpression)  // Assigns the value 12 to the variable total.
var totality = eval("...surrounded by acres of clams.");
```

Consult the [language reference](#) for more information about these and other built-in functions.

**Creating Your Own Functions**

You can create your own functions and use them where you need them. A function definition consists of a function statement and a block of JScript statements.

The `checkTriplet` function in the following example takes as its arguments the lengths of the sides of a triangle, and calculates from them whether the triangle is a right triangle by checking whether the three numbers constitute a Pythagorean triplet. (The square of the length of the hypotenuse of a right triangle is equal to the sum of the squares of the lengths of the other two sides.) The `checkTriplet` function calls one of two other functions to make
the actual test.

Notice the use of a very small number ("epsilon") as a testing variable in the floating-point version of the test. Because of uncertainties and roundoff errors in floating-point calculations, it is not practical to make a direct test of whether the square of the hypotenuse is equal to the sum of the squares of the other two sides unless all three values in question are known to be integers. Because a direct test is more accurate, the code in this example determines whether it is appropriate and, if it is, uses it.

```javascript
var epsilon = 0.0000000000001; // Some very small number to use for testing.
var triplet = false;

function integerCheck(a, b, c) { // The test function for integers.
    if ( (a*a) == ((b*b) + (c*c)) ) { // The test itself.
        triplet = true;
    }
}
// End of the integer checking function.

function floatCheck(a, b, c) { // The test function for floating-point numbers.
    var theCheck = ((a*a) - ((b*b) + (c*c))) // Make the test number.
    if (theCheck < 0) { // The test requires the absolute value.
        theCheck *= -1;
    }
    if (epsilon > theCheck) { // If it's as close as that, it's a match.
        triplet = true;
    }
}
// End of the floating-point check function.

function checkTriplet(a, b, c) { // The triplet checker.
    // The actual test.

    // Notice the use of a very small number ("epsilon") as a testing variable in
    // the floating-point version of the test. Because of uncertainties and roundoff
    // errors in floating-point calculations, it is not practical to make a direct test
    // of whether the square of the hypotenuse is equal to the sum of the squares
    // of the other two sides unless all three values in question are known to be
    // integers. Because a direct test is more accurate, the code in this example
determines whether it is appropriate and, if it is, uses it.

    var epsilon = 0.0000000000001; // Some very small number to test against.
    var triplet = false;

    function integerCheck(a, b, c) { // The test function for integers.
        if ( (a*a) == ((b*b) + (c*c)) ) { // The test itself.
            triplet = true;
        }
    }
    // End of the integer checking function.

    function floatCheck(a, b, c) { // The test function for floating-point numbers.
        var theCheck = ((a*a) - ((b*b) + (c*c))) // Make the test number.
        if (theCheck < 0) { // The test requires the absolute value.
            theCheck *= -1;
        }
        if (epsilon > theCheck) { // If it's as close as that, it's a match.
            triplet = true;
        }
    }
    // End of the floating-point check function.

    function checkTriplet(a, b, c) { // The triplet checker.
var d = 0; // Create a temporary holding bin.
    if (c > b) { // If c > b, swap them.
        d = c;
        c = b;
        b = d;
    } // If not, ignore them.
    if (b > a) { // If b > a, swap them.
        d = b;
        b = a;
        a = d;
    } // If not, ignore them.

    // Side "a" is now the hypotenuse, if there is one.
    if (((a%1 == 0) && ((b%1 == 0) && ((c%1 == 0)
    integerCheck(a, b, c); // If so, use the precise check
}    else
    floatCheck(a, b, c); // If not, get as close as is reasonab
} // End of the triplet check function.

    // The next three statements assign sample values for t
    var sideA = 5;
    var sideB = 5;
    var sideC = Math.sqrt(50);

    checkTriplet(sideA, sideB, sideC); // Call the function
What Are Objects?

In Microsoft JScript, objects are, essentially, collections of properties and methods. A method is a function that is a member of an object, and a property is a value or set of values (in the form of an array or object) that is a member of an object. JScript supports three kinds of objects: intrinsic objects, objects you create, and browser objects, which are covered elsewhere.

Objects as Arrays

In JScript, objects and arrays are handled identically. You can refer to any of the members of an object (its properties and methods) either by name (using the name of the object, followed by a period, followed by the name of the property) or by its array subscript index. Subscript numbering in JScript begins with 0. For convenience, the subscript can also be referred to by its name.

Thus, a property can be referred to in several ways. All of the following statements are equivalent.

```javascript
theWidth = spaghetti.width;
theWidth = spaghetti[3];  // [3] is the "width" index.
theWidth = spaghetti["width"];
```

While it is possible to use brackets to refer to a property by its
numeric index, it is not possible to use the dot (.) convention with index numbers. The following statement generates an error.

```javascript
theWidth = spaghetti.3;
```

When an object has another object as a property, the naming convention extends in a straightforward way.

```javascript
var init4 = todotoday.shoppingList[3].substring(0,1);
```

The fact that objects can have other objects as properties lets you generate arrays with more than one subscript, which are not directly supported. The following code creates a multiplication table for values from 0 times 0 through 16 times 16.

```javascript
var multTable = new Array(17); // Make the shell that will become the table.
for (var j = 0; j < multTable.length; j++) {
    // Prepare to fill it with rows.
    var aRow = new Array(17); // Create a row.
    for (var i = 0; i < aRow.length; i++) {
        // Prepare to make an element.
        aRow[i] = (i + " times " + j + " = " + i*j); // Make and place one value.
    }
    multTable[j] = aRow; // Put the filled row into the table.
}
```

To refer to one of the elements of an array of this kind, use multiple sets of brackets.

```javascript
var multiply3x7 = multTable[3][7];
```

The following statement generates an error.

```javascript
var multiply3x7 = multTable[3, 7];
```
JScript has a number of reserved keywords. These words come in three types: JScript reserved keywords, future reserved words, and words to avoid.

<table>
<thead>
<tr>
<th>JScript Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>break</td>
</tr>
<tr>
<td>false</td>
</tr>
<tr>
<td>in</td>
</tr>
<tr>
<td>this</td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td>continue</td>
</tr>
<tr>
<td>for</td>
</tr>
<tr>
<td>new</td>
</tr>
<tr>
<td>true</td>
</tr>
<tr>
<td>while</td>
</tr>
<tr>
<td>delete</td>
</tr>
<tr>
<td>function</td>
</tr>
<tr>
<td>null</td>
</tr>
<tr>
<td>typeof</td>
</tr>
<tr>
<td>with</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>if</td>
</tr>
<tr>
<td>return</td>
</tr>
<tr>
<td>var</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>JScript Future Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>case</td>
</tr>
<tr>
<td>debugger</td>
</tr>
<tr>
<td>export</td>
</tr>
<tr>
<td>super</td>
</tr>
<tr>
<td>catch</td>
</tr>
<tr>
<td>default</td>
</tr>
<tr>
<td>extends</td>
</tr>
<tr>
<td>switch</td>
</tr>
<tr>
<td>class</td>
</tr>
<tr>
<td>do</td>
</tr>
<tr>
<td>finally</td>
</tr>
<tr>
<td>throw</td>
</tr>
<tr>
<td>const</td>
</tr>
<tr>
<td>enum</td>
</tr>
<tr>
<td>import</td>
</tr>
<tr>
<td>try</td>
</tr>
</tbody>
</table>

The words to avoid are any that are already the names of intrinsic JScript objects or functions. Words like String or parseInt are included in this.

Using any of the keywords from the first two categories causes a compilation error when your script is first loaded. Using a reserved word from the third set can cause odd behavior problems if you attempt to use both your variable and the original entity of the same name in the same script. For example, the following script does not do quite what you think it should:

```javascript
var String;
```
var text = new String("This is a string object");

In this case, you get an error saying that String is not an object. Many cases of using a pre-existing identifier aren't this obvious.
Recursion is an important programming technique. It's used to have a function call itself from within itself. One handy example is the calculation of factorials. The factorials of 0 and 1 are both defined specifically to be 1. The factorials of larger numbers are calculated by multiplying 1 * 2 * ..., incrementing by 1 until you reach the number for which you're calculating the factorial.

The following paragraph is a function, defined in words, that calculates a factorial.

"If the number is less than zero, reject it. If it isn't an integer, round it down to the next integer. If the number is zero or one, its factorial is one. If the number is larger than one, multiply it by the factorial of the next smaller number."

To calculate the factorial of any number that is larger than 1, you need to calculate the factorial of at least one other number. The function you use to do that is the function you're in the middle of already; the function must call itself for the next smaller number, before it can execute on the current number. This is an example of recursion.

Clearly, there is a way to get in trouble here. You can easily create a recursive function that doesn't ever get to a definite result, and cannot reach an endpoint. Such a recursion causes the computer to execute a so-called "infinite" loop. Here's an example: omit the first rule (the one about negative numbers) from the verbal description of calculating a factorial, and try to calculate the factorial of any negative number. This fails, because in order to calculate the factorial of, say, -24 you first have to calculate the factorial of -25; but in order to do that you first have to calculate the factorial of -26; and so on. Obviously, this never reaches a stopping place.
Thus, it is extremely important to design recursive functions with great care. If you even suspect that there's any chance of an infinite recursion, you can have the function count the number of times it calls itself. If the function calls itself too many times, however many you decide that should be, it automatically quits.

Here's the factorial function again, this time written in JScript code.

```javascript
function factorial(aNumber) {
    aNumber = Math.floor(aNumber); // If the number is not an integer, round it down.
    if (aNumber < 0) { // If the number is less than zero, reject it.
        return "not a defined quantity";
    }
    if ((aNumber == 0) || (aNumber == 1)) { // If the number is 0 or 1, its factorial is 1.
        return 1;
    }
    else return (aNumber * factorial(aNumber - 1)); // Otherwise, recurse until done.
}
```
Microsoft JScript has two scopes: global and local. If you declare a variable outside of any function definition, it is a global variable, and its value is accessible and modifiable throughout your program. If you declare a variable inside of a function definition, that variable is local. It is created and destroyed every time the function is executed; it cannot be accessed by anything outside the function.

A local variable can have the same name as a global variable, but it is entirely distinct and separate. Consequently, changing the value of one variable has no effect on the other. Inside the function in which the local variable is declared, only the local version has meaning.

```javascript
var aCentaur = "a horse with rider,"; // Global definition

// JScript code, omitted for brevity.
function antiquities() // A local aCentaur variable is declared.
{

    // JScript code, omitted for brevity.
    var aCentaur = "A centaur is probably a mounted Scyth

    // JScript code, omitted for brevity.
    aCentaur += ", misreported; that is, "; // Adds to the

    // JScript code, omitted for brevity.
} // End of the function.
```
var nothinginparticular = antiquities();
aCentaur += " as seen from a distance by a naive innocent.";

/*
 Within the function, the variable contains "A centaur is probably a mounted Scythian warrior, misreported; that is, "; outside the function, the variable contains the rest of the sentence: "a horse with rider, as seen from a distance by a naive innocent."
 */

It's important to note that variables act as if they were declared at the beginning of whatever scope they exist in. Sometimes this results in unexpected behaviors.

```javascript
var aNumber = 100;
var withAdditive = 0;

withAdditive += aNumber; // withAdditive is now 100
tweak();
withAdditive += aNumber; // withAdditive is now 200

function tweak() {
  var newThing = 0; // Explicit declaration of the newThing variable.
  // The next statement, if it were not commented out, would assign the value 42 to the local
  // newThing = aNumber;
  // The next statement assigns the value 42 to the local
  aNumber = 42;
  if (false) {
```
The statement that is commented out attempts to assign the value of the local variable aNumber to the local variable newThing. It fails, despite the fact that a local aNumber variable is defined elsewhere in the function, and therefore exists throughout. The aNumber variable does not have any assigned value at the point where this statement occurs in the code, and is thus undefined.
In Microsoft JScript, how data is handled depends on its data type.

**By Value vs. By Reference**

Numbers and Boolean values (true and false) are copied, passed, and compared by value. When you copy or pass by value, you allocate a space in computer memory and put the value of the original into it. If you then change the original, the copy is not affected (and vice versa), because the two are separate entities.

Objects, arrays, and functions are copied, passed, and compared by reference under most circumstances. When you copy or pass by reference, you essentially create a pointer to the original item, and use the pointer as if it were a copy. If you then change the original, you change both the original and the copy (and vice versa). There is really only one entity; the "copy" is not actually a copy, it's just another reference to the data.

**Note** You can change this behavior for objects and arrays by specifying the assign() method for them.

Last, strings are copied and passed by reference, but are compared by value.

**Note** Because of the way the ASCII and ANSI character sets are constructed, capital letters precede lowercase ones in sequence order. For example, "Zoo" compares as less than
"aardvark."

---

Passing Parameters to Functions

When you pass a parameter to a function by value, you are making a separate copy of that parameter, a copy that exists only inside the function. If, on the other hand, you pass a parameter by reference, and the function changes the value of that parameter, it is changed everywhere in the script.

Testing Data

When you perform a test by value, you compare two distinct items to see whether they are equal to each other. Usually, this comparison is performed on a byte-by-byte basis. When you test by reference, you are checking to see whether two items are pointers to a single original item. If they are, then they compare as equal; if not, even if they contain the exact same values, byte-for-byte, they compare as unequal.

Copying and passing strings by reference saves memory; but because you cannot change strings once they are created, it becomes possible to compare them by value. This lets you test whether two strings have the same content even if one was generated entirely separately from the other.
Array Indexing

Arrays in JScript are *sparse*. That is, if you have an array with three elements that are numbered 0, 1, and 2, you can create element 50 without worrying about elements 3 through 49. If the array has an automatic length variable (see *Intrinsic Objects* for an explanation of automatic monitoring of array length), the length variable is set to 51, rather than to 4. You can certainly create arrays in which there are no gaps in the numbering of elements, but you aren't required to. In fact, in JScript, your arrays don't have to have numbered subscripts at all.

In JScript, objects and arrays are essentially identical to each other. The real difference is not in the data, but rather in the way you address the members of an array or the properties and methods of an object.

Addressing Arrays

There are two main ways to address the members of an array. Ordinarily, you address arrays by using brackets. The brackets enclose either a numeric value or an *expression* that evaluates to a nonnegative integer. The following example assumes that the *entryNum* variable is defined and assigned a value elsewhere in the script.

```javascript
theListing = addressBook[entryNum];
theFirstLine = theListing[1];
```

This method of addressing is equivalent to the method for
addressing objects, though in object addressing, what follows the period must be the name of an actual property. If there is no such property, your code generates an error.

The second way to address an array is to make an object/array that contains properties that are numbered, and then generate the numbers in a loop. The following example generates two arrays, one for the name and one for the address, from a listing in \textit{addressBook}. Each of these contains four properties. An instance of \texttt{theName}, for example, built from the \texttt{Name1} through \texttt{Name4} properties of \texttt{theListing}, might contain "G." "Edward" "Heatherington" "IV", or "George" "" "Sand" "".

\begin{verbatim}
theListing = addressBook[entryNum];
for (i = 1; i < 4; i++) {
  theName[i] = theListing["Name" + i];
  theAddress[i] = theListing["Address" + i];
}
\end{verbatim}

While this particular instance is short, and could easily have been written in the "dot" style of notation, (that is, addressing \texttt{theListing}, \texttt{theName}, and \texttt{theAddress} as objects rather than as arrays), that is not always possible. Sometimes the particular property may not exist until run time, or there may be no way to know which one it will be in advance. For example, if the \texttt{addressBook} array were arranged by last name instead of by numbered listings, the user would probably be entering names "on the fly," while the script is running, to look people up. The following example assumes the existence of appropriate function definitions elsewhere in the script.

\begin{verbatim}
theListing = addressBook[getName];
\end{verbatim}
theIndivListing = theListing[getFirstName();

This is *associative* addressing of the array, that is, addressing by means of fully arbitrary strings. Objects in JScript are actually associative arrays. Although you can (and frequently do) use the "dot" style of addressing, you are not by any means required to. Because the members of any JScript object can be accessed using array notation, a JScript object can be used as an associative array.

The following code creates and initializes the most familiar form of an array:

```javascript
var myArray = new Array("Athens", "Belgrade", "Cairo");
```

Each element of this array is addressed using its element number; in this case 0, 1, or 2. Using the `for...in` statement, the array can be iterated starting at 0 and ending at 2. For example:

```javascript
for (key in myArray)
    response.write("Element value is ' + MyArray[key] + '<BR>);
```

The following code creates and initializes an associative array containing three elements:

```javascript
var MyArray = {"a": "Athens", "b": "Belgrade", "c": "Cairo"};
```

In this array, elements are addressed using the key strings("a", "b", or "c") instead of an array element number (0, 1, or 2). This
allows you to create and use arrays with more intuitive addressing schemes. The same `for...in` statement code shown above can be used to iterate this array as well.
Using Constructors to Create Objects

In Microsoft JScript, you use constructors to create and build a class of objects. You invoke a constructor with the `new` statement. It returns whatever it constructs.

The special case Function constructor lets you create functions that are anonymous. An anonymous function is one that does not have a name. You can use the Function constructor to build a function "on the fly", for example, as one of the instructions inside another function. Such a function, which is only called from the one location, doesn't need a name.

In the following example, such an anonymous function generates one line of a "name-and-email-address" listing. It checks the value of the `firstNameFirst` variable to decide whether to put the first name or the last name first, and the value of the `emailNameFirst` variable to decide whether to put the name or the email address first. The example assumes that the values of `firstNameFirst` and `emailNameFirst` are set elsewhere.

```javascript
for (j = 1; j < addressList[length]; j++)
{
    oneListingLine = new Function(emailNameFirst, firstNameFirst, addressList, j,
    
    if(firstNameFirst)
    {
        theName=(addressList[j].firstName + addressList[j].lastName);
    },);
```
if (emailNameFirst)
{
    theListing = addressList[j].emailName + ":\t" + theName
} else theListing = theName + "
document.write(oneListingLine + "

Writing Constructors

To write your own constructors, you use the this keyword within the constructor to refer to the newly-created object. The constructor initializes the object.

Though the constructor in the next example starts at an index of 0, this is not required. You can start with a first index of 1 if, for example, you want a parameter that indicates the actual number of indexes of the array or object. In the example, it's called extent to distinguish it from the automatically maintained length parameter of the built-in Array( ) object. If you write code that adds properties to the array, you have to update the extent parameter (or your equivalent) because this parameter is not maintained by JScript. Notice that even this extremely simple example uses both object (dot) and array (bracket) notation styles to refer to the current object.

function MakeStringArray(length)
this.extent = length;
for (iNum = 0; iNum < length; i++)
    this[iNum] = "";
Using Prototypes to Create Objects

When you write an object definition, you can use prototype properties to create properties that are held in common by all objects that are generated by the definition. Prototype properties are copied by reference into each object of a class, so they have the same value for all objects in the class. However, you can change the value of a prototype property in one object, and the new value overrides the default, but only in that one instance. Other objects that are members of the class are not affected by the change.

Using this principle, you can define additional properties for objects that are part of the JScript language, which all have prototypes. For example, if you want a special constant for a calculation, and the constant is not among those provided in the Math object, you can define it yourself and then assign it their respective object prototypes, or the prototype property of your object class.

Math.prototype.Avogadros = 6.0232e23;

function howManyMolecules(wtGr, molWt) {
  return ((wtGrams/molWt)*Math.prototype.Avogadros);
document.write("There are " + how ("What's the molecular weight?",0) " molecules in that amount.");

Perhaps more to the point, you can define a function, assign it to String.prototype as a method, and use it on any string anywhere in your script. The following example assumes the existence of a Periodic Chart array called "theElements", defined elsewhere in the script, which contains symbols for the elements, their names, their atomic weights, and other relevant information about them.

```javascript
function atomName(theSymbol) {
    return(theElements[theSymbol].fullName);
}

String.prototype.atomName = atomName;

function decodeFormula(theFormula) {
    var theCurrentPiece = "";
```
```javascript
var theDecodedFormula = "";
for (i = 1; i = theFormula.length ; i++)
if (theFormtheCurrentPiece
// Code statements to separate the formula
// Loop through the formula array and assemble the decoded string. Each term is:
theDecodedFormula += formula[n].number
theDecodedFormula += " ";
theDecodedFormula += formula[n].symbol.prototype.atomName;
theDecodedFormula += " ";
// End of loop.

return theDecodedFormula;
}

decodeFormula(window.prompt("Formula?","Al2O3"));
```
Special Characters

JScript provides special characters that allow you to include in strings some characters you can't type directly. Each of these characters begins with a backslash. The backslash is an *escape* character you use to inform the JScript interpreter that the next character is special.

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\b</code></td>
<td>Backspace</td>
</tr>
<tr>
<td><code>\f</code></td>
<td>Form feed</td>
</tr>
<tr>
<td><code>\n</code></td>
<td>Line feed (newline)</td>
</tr>
<tr>
<td><code>\r</code></td>
<td>Carriage return</td>
</tr>
<tr>
<td><code>\t</code></td>
<td>Horizontal tab (Ctrl-I)</td>
</tr>
<tr>
<td><code>'</code></td>
<td>Single quotation mark</td>
</tr>
<tr>
<td><code>&quot;</code></td>
<td>Double quotation mark</td>
</tr>
<tr>
<td><code>\</code></td>
<td>Backslash</td>
</tr>
</tbody>
</table>

Notice that because the backslash itself is used as the escape character, you cannot directly type one in your script. If you want to write a backslash, you must type two of them together (`\`).

```javascript
document.write('The image path is C:\webstuff\mypage\gifs\garden.gif.');
document.write('The caption reads, “After the snow of ’97. Grandma’s house is covered.”');
```
You can use these escape sequences to control formatting of text inside `<PRE>` and `<XMP>` tags and, to some extent, inside alert, prompt, and confirm message boxes.
There are places in any programming language where you can get caught if you are not careful, and every language has specific surprises in it. Take, for example, the `null` value: The one in Microsoft JScript behaves differently than the `null` value in the C or C++ languages.

Here are some of the trouble areas that you may run into as you write JScript scripts.

**Syntax Errors**

Because syntax is much more rigid in programming languages than in natural languages, it is important to pay strict attention to detail when you write scripts. If, for example, you mean for a particular parameter to be a string, you will run into trouble if you forget to enclose it in quotation marks when you type it.

**Order of Script Interpretation**

JScript interpretation is part of the your Web browser's HTML parsing process. So, if you place a script inside the `<HEAD>` tag in a document, it is interpreted before any of the `<BODY>` tag is examined. If you have objects that are created in the `<BODY>` tag, they do not exist at the time the `<HEAD>` is being parsed, and cannot be manipulated by the script.

**Automatic Type Coercion**

JScript is a loosely typed language with automatic coercion. Consequently, despite the fact that values having different types
are not equal, the expressions in the following example evaluate to **true**.

"100" == 100  
false == 0

**Operator Precedence**

When a particular operation is performed during the evaluation of an expression has more to do with operator precedence than with the location of the expression. Thus, in the following example, multiplication is performed before subtraction, even though the subtraction appears first in the expression.

```
theRadius = aPerimeterPoint - theCenterpoint * 
```

**Using for...in Loops with Objects**

When you step through the properties of an object with a **for...in** loop, you cannot necessarily predict or control the order in which the fields of the object are assigned to the loop counter variable. Moreover, the order may be different in different implementations of the language.

**with Keyword**

The **with** statement is convenient for addressing properties that already exist in a specified object, but cannot be used to add properties to an object. To create new properties in an object, you must refer to the object specifically.

**this Keyword**
Although you use the *this* keyword inside the definition of an object, to refer to the object itself, you cannot ordinarily use *this* or similar keywords to refer to the currently executing function when that function is not an object definition. You can, if the function is to be assigned to an object as a method, use the *this* keyword within the function, to refer to the object.

**Writing a Script That Writes a Script**

The `<SCRIPT>` tag terminates the current script if the interpreter encounters it. To display "</SCRIPT>" itself, rewrite this as at least two strings, for example, "</SCR" and "IPT>", which you can then concatenate together in the statement that writes them out.

**Implicit Window References**

Because more than one window can be open at a time, any window reference that is implicit is taken to point to the current window. For other windows, you must use an explicit reference.
Microsoft JScript provides two ways to display data directly in your browser. You can use the `write()` and `writeln()`, which are methods of the `document` object. You can also display information in forms within the browser, and in `alert`, `prompt`, and `confirm` message boxes.

Using `document.write()` and `document.writeln()`

The most common way to display information is the `write()` method of the `document` object. It takes one argument, a string, which it displays in the browser. The string can be either plain text or HTML.

Strings can be enclosed in either single or double quotation marks. This lets you quote something that contains quote marks or apostrophes.

```javascript
document.write("Pi is approximately\n\ndocument.write(\);
```

**Tip** The following simple function is a way around having something to appear in the browser window. This function `write` is undefined, but does let you issue the command "write".

```javascript
function w(m) { // Write function
```
m = "" + m + ";";  // Make sure that
if ("undefined" != m) {   // Test for
document.write(m);
}
document.write("<br>");
}

w('<IMG SRC="horse.gif">');
w();
w("This is an engraving of a horse.");
w();

The writeln() method is almost identical to the write() method, except that it appends a newline character to whatever string you provide. In HTML this ordinarily results only in a space after your item; but if you're using <PRE> and <XMP> tags, the newline character is interpreted literally and the browser displays it.

When you call the write() method, it opens and clears the document if the document is not in the process of being opened and parsed when the write() method is called, so it can be dangerous. The example shows a script that is intended to display the time once a minute, but fails to do so after the first time because it clears itself in the process.

<HTML>
<HEAD>
function singOut() {
    var theMoment = new Date();
    var theHour = theMoment.getHours();
    var theMinute = theMoment.getMinutes();
    var theDisplacement = (theMoment.getTimezoneOffset() / 60);
    theHour -= theDisplacement;
    if (theHour > 23) {
        theHour -= 24
    }
    document.write(theHour + " hours, 
    window.setTimeout("singOut();", 60000);
}
</SCRIPT>
</HEAD>
</BODY>
<SCRIPT>
singOut();
</SCRIPT>
</BODY>
</HTML>

If you use the `alert()` method of the window object instead of `document.write()`, the script works.

```javascript
window.alert(theHour + " hours, ", window.setTimeout("singOut();", 60000);
}
```

Clearing the Current Document

The `clear()` method of the `document` object empties the current document. This method also clears your script (along with the rest of the document), so be very careful how and when you use it.

```javascript
document.clear();
```
Welcome to the Scripting Run-Time Library Reference

These handy blocks of information will help you explore the many different parts of the Scripting Run-Time Library.

You'll find all the parts of the Scripting Run-Time Library listed alphabetically under the Alphabetic Keyword List. But if you want to examine just one category, say, objects, each language category has its own, more compact section.

How's it work? Click on one of the headings to the left to display a list of items contained in that category. From this list, select the topic that you want to view. Once you've opened that topic, you can easily link to other related sections.

So, go ahead and take a look! Study some statements, mull over the methods, or figure out a few functions. You'll see just how versatile the Scripting Run-Time Library can be!
| Microsoft Scripting Run-Time Features | List of features currently in Microsoft Scripting Run-Time Library. |

© 2000 Microsoft Corporation. All rights reserved.
The FileSystemObject Object Model

When writing scripts for Active Server Pages, the Windows Scripting Host, or other applications where scripting can be used, it's often important to add, move, change, create, or delete folders (directories) and files on the Web server. It may also be necessary to get information about and manipulate drives attached to the Web server.

Scripting allows you to process drives, folders, and files using the FileSystemObject (FSO) object model, which is explained in the following sections:

- [Introduction to the FileSystemObject and the Scripting Run-Time Library Reference](#)
- [FileSystemObject Objects](#)
- [Programming the FileSystemObject](#)
- [Working with Drives and Folders](#)
- [Working with Files](#)
- [FileSystemObject Sample Code](#)
The **FileSystemObject** (FSO) object model allows you to use the familiar *object.method* syntax with a rich set of properties, methods, and events to process folders and files.

Use this object-based tool with:

- HTML to create Web pages
- Windows Scripting Host to create batch files for Microsoft Windows
- Script Control to provide a scripting capability to applications developed in other languages

Because use of the FSO on the client side raises serious security issues about providing potentially unwelcome access to a client's local file system, this documentation assumes use of the FSO object model to create scripts executed by Internet Web pages on the server side. Since the server side is used, the Internet Explorer default security settings do not allow client-side use of the **FileSystemObject** object. Overriding those defaults could subject a local computer to unwelcome access to the file system, which could result in total destruction of the file system's integrity, causing loss of data, or worse.
The FSO object model gives your server-side applications the ability to create, alter, move, and delete folders, or to detect if particular folders exist, and if so, where. You can also find out information about folders, such as their names, the date they were created or last modified, and so forth.

The FSO object model also makes it easy to process files. When processing files, the primary goal is to store data in a space- and resource-efficient, easy-to-access format. You need to be able to create files, insert and change the data, and output (read) the data. Since storing data in a database, such as Access or SQL Server, adds a significant amount of overhead to your application, storing your data in a binary or text file may be the most efficient solution. You may prefer not to have this overhead, or your data access requirements may not require all the extra features associated with a full-featured database.

The FSO object model, which is contained in the Scripting type library (Scrrun.dll), supports text file creation and manipulation through the TextStream object. Although it does not yet support the creation or manipulation of binary files, future support of binary files is planned.
The **FileSystemObject** (FSO) object model contains the following objects and collections.

<table>
<thead>
<tr>
<th>Object/Collection</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FileSystemObject</strong></td>
<td>Main object. Contains methods and properties that allow you to create, delete, gain information about, and generally manipulate drives, folders, and files. Many of the methods associated with this object duplicate those in other FSO objects; they are provided for convenience.</td>
</tr>
<tr>
<td>Drive</td>
<td>Object. Contains methods and properties that allow you to gather information about a drive attached to the system, such as its share name and how much room is available. Note that a &quot;drive&quot; isn't necessarily a</td>
</tr>
</tbody>
</table>
hard disk, but can be a CD-ROM drive, a RAM disk, and so forth. A drive doesn't need to be physically attached to the system; it can be also be logically connected through a network.

<table>
<thead>
<tr>
<th>Drives</th>
<th>Collection. Provides a list of the drives attached to the system, either physically or logically. The Drives collection includes all drives, regardless of type. Removable-media drives need not have media inserted for them to appear in this collection.</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>Object. Contains methods and properties that allow you to create, delete, or move a file. Also allows you to query the system for a file name, path, and various other properties.</td>
</tr>
<tr>
<td>Files</td>
<td>Collection. Provides a list of all files contained within</td>
</tr>
<tr>
<td><strong>Folder</strong></td>
<td>a folder.</td>
</tr>
<tr>
<td>------------</td>
<td>----------</td>
</tr>
<tr>
<td><strong>Folder Object.</strong></td>
<td>Contains methods and properties that allow you to create, delete, or move folders. Also allows you to query the system for folder names, paths, and various other properties.</td>
</tr>
<tr>
<td><strong>Folders</strong></td>
<td>Collection. Provides a list of all the folders within a Folder.</td>
</tr>
<tr>
<td><strong>TextStream</strong></td>
<td>Object. Allows you to read and write text files.</td>
</tr>
</tbody>
</table>
To program with the **FileSystemObject** (FSO) object model:

- Use the **CreateObject** method to create a **FileSystemObject** object.
- Use the appropriate method on the newly created object.
- Access the object's properties.

The FSO object model is contained in the Scripting **type library**, which is located in the Scrrun.dll file. Therefore, you must have Scrrun.dll in the appropriate system directory on your Web server to use the FSO object model.

### Creating a FileSystemObject Object

First, create a **FileSystemObject** object by using the **CreateObject** method. In VBScript, use the following code to create an instance of the **FileSystemObject**:

```vbscript
Dim fso
Set fso = CreateObject("Scripting.FileSystemObject")
```

This [sample code](#) demonstrates how to create an instance of the **FileSystemObject**.

In JScript, use this code to do the same thing:
var fso;
fso = new ActiveXObject("Scripting.FileSystemObject");

In both of these examples, **Scripting** is the name of the type library and **FileSystemObject** is the name of the object that you want to create. You can create only one instance of the **FileSystemObject** object, regardless of how many times you try to create another.

**Using the Appropriate Method**

Second, use the appropriate method of the **FileSystemObject** object. For example, to create a new object, use either **CreateTextFile** or **CreateFolder** (the FSO object model doesn't support the creation or deletion of drives).

To delete objects, use the **DeleteFile** and **DeleteFolder** methods of the **FileSystemObject** object, or the **Delete** method of the **File** and **Folder** objects. You can also copy and move files and folders, by using the appropriate methods.

---

**Note** Some functionality in the **FileSystemObject** object model is redundant. For example, you can copy a file using either the **CopyFile** method of the **FileSystemObject** object, or you can use the **Copy** method of the **File** object. The methods work the same; both exist to offer programming flexibility.

---

**Accessing Existing Drives, Files, and Folders**

To gain access to an existing drive, file, or folder, use the appropriate "get" method of the **FileSystemObject** object:
• GetDrive
• GetFolder
• GetFile

To gain access to an existing file in VBScript:

    Dim fso, f1
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f1 = fso.GetFile("c:\test.txt")

To do the same thing in JScript, use the following code:

    var fso, f1;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    f1 = fso.GetFile("c:\\test.txt");

Do not use the "get" methods for newly created objects, since the "create" functions already return a handle to that object. For example, if you create a new folder using the CreateFolder method, don't use the GetFolder method to access its properties, such as Name, Path, Size, and so forth. Just set a variable to the CreateFolder function to gain a handle to the newly created folder, then access its properties, methods, and events. To do this in VBScript, use the following code:

    Sub CreateFolder
        Dim fso, fldr
        Set fso = CreateObject("Scripting.FileSystemObject'
        Set fldr = fso.CreateFolder("C:\MyTest")
        Response.Write "Created folder: " & fldr.Name
function CreateFolder()
{
    var fso, fldr;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    fldr = fso.CreateFolder("C:\MyTest");
    Response.Write("Created folder: " + fldr.Name);
}

Accessing the Object’s Properties

Once you have a handle to an object, you can access its properties. For example, to get the name of a particular folder, first create an instance of the object, then get a handle to it with the appropriate method (in this case, the GetFolder method, since the folder already exists).

Use this code to get a handle to the GetFolder method in VBScript:

    Set fldr = fso.GetFolder("c:\")

To do the same thing in JScript, use the following code:

    var fldr = fso.GetFolder("c:\\");

Now that you have a handle to a Folder object, you can check its Name property. Use the following code to check this in VBScript:
Response.Write "Folder name is: " & fldr.Name

To check a **Name** property in JScript, use this syntax:

```javascript
Response.Write("Folder name is: " + fldr.Name);
```

To find out the last time a file was modified, use the following VBScript syntax:

```vbscript
Dim fso, f1
Set fso = CreateObject("Scripting.FileSystemObject")
' Get a File object to query.
Set f1 = fso.GetFile("c:\detlog.txt")
' Print information.
Response.Write "File last modified: " & f1.DateLastModified
```

To find out the same thing in JScript, use this code:

```javascript
var fso, f1;
fso = new ActiveXObject("Scripting.FileSystemObject");
// Get a File object to query.
f1 = fso.getFile("c:\\detlog.txt");
// Print information.
Response.Write("File last modified: " + f1.DateLastModified);
```
With the **FileSystemObject** (FSO) object model, you can work with drives and folders programmatically just as you can in the Windows Explorer interactively. You can copy and move folders, get information about drives and folders, and so forth.

### Getting Information About Drives

The **Drive** object allows you to gain information about the various drives attached to a system, either physically or over a network. Its properties allow you to obtain information about:

- The total size of the drive in bytes (**TotalSize** property)
- How much space is available on the drive in bytes (**AvailableSpace** or **FreeSpace** properties)
- What letter is assigned to the drive (**DriveLetter** property)
- What type of drive it is, such as removable, fixed, network, CD-ROM, or RAM disk (**DriveType** property)
- The drive's serial number (**SerialNumber** property)
- The type of file system the drive uses, such as FAT, FAT32, NTFS, and so forth (**FileSystem** property)
- Whether a drive is available for use (**IsReady** property)
- The name of the share and/or volume (**ShareName** and **VolumeName** properties)
• The path or root folder of the drive (Path and RootFolder properties)

View the sample code to see how these properties are used in FileSystemObject.

Example Usage of the Drive Object

Use the Drive object to gather information about a drive. You won't see a reference to an actual Drive object in the following code; instead, use the GetDrive method to get a reference to an existing Drive object (in this case, drv).

The following example demonstrates how to use the Drive object in VBScript:

```vbnet
Sub ShowDriveInfo(drvPath)
    Dim fso, drv, s
    Set fso = CreateObject("Scripting.FileSystemObject'
    Set drv = fso.GetDrive(fso.GetDriveName(drvPath))
    s = "Drive " & UCase(drvPath) & " - "
    s = s & drv.VolumeName & "<br>"
    s = s & "Total Space: " & FormatNumber(drv.TotalSize / 1024, 0) & " Kb" & "<br>"
    s = s & "Free Space: " & FormatNumber(drv.FreeSpace / 1024, 0) & " Kb" & "<br>"
    Response.Write s
End Sub
```

The following code illustrates the same functionality in JScript:
function ShowDriveInfo1(drvPath)
{
    var fso, drv, s = "";
    fso = new ActiveXObject("Scripting.FileSystemObject");
drv = fso.GetDrive(fso.GetDriveName(drvPath));
s += "Drive " + drvPath.toUpperCase() + " - ";
s += drv.VolumeName + "<br>";
s += "Total Space: " + drv.TotalSize / 1024;
s += " Kb" + "<br>";
s += "Free Space: " + drv.FreeSpace / 1024;
s += " Kb" + "<br>";
    Response.Write(s);
}

Working with Folders

Common folder tasks and the methods for performing them are described in the following table.

<table>
<thead>
<tr>
<th>Task</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a folder.</td>
<td>FileSystemObject.CreateFolder</td>
</tr>
<tr>
<td>Delete a folder.</td>
<td>Folder.Delete or FileSystemObject.DeleteFolder</td>
</tr>
<tr>
<td>Move a folder.</td>
<td>Folder.Move or FileSystemObject.MoveFolder</td>
</tr>
<tr>
<td>Copy a folder.</td>
<td>Folder.Copy or FileSystemObject.CopyFolder</td>
</tr>
<tr>
<td>Retrieve the name of a folder.</td>
<td>Folder.Name</td>
</tr>
<tr>
<td>Find out if a folder</td>
<td></td>
</tr>
<tr>
<td>Action</td>
<td>Code</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>exists on a drive.</td>
<td>FileSystemObject.FolderExists</td>
</tr>
<tr>
<td>Get an instance of an existing Folder object.</td>
<td>FileSystemObject.GetFolder</td>
</tr>
<tr>
<td>Find out the name of a folder's parent folder.</td>
<td>FileSystemObject.GetParentFolderName</td>
</tr>
<tr>
<td>Find out the path of system folders.</td>
<td>FileSystemObject.GetSpecialFolder</td>
</tr>
</tbody>
</table>

View the [sample code](#) to see how many of these methods and properties are used in **FileSystemObject**.

The following example demonstrates how to use the **Folder** and **FileSystemObject** objects to manipulate folders and gain information about them in VBScript:

```vbnet
Sub ShowFolderInfo()
    Dim fso, fldr, s
    ' Get instance of FileSystemObject.
    Set fso = CreateObject("Scripting.FileSystemObject"
    ' Get Drive object.
    Set fldr = fso.GetFolder("c:")
    ' Print parent folder name.
    Response.Write "Parent folder name is: " & fldr & "<br>
    ' Print drive name.
    Response.Write "Contained on drive " & fldr.Drive & "<br>
    ' Print root file name.
    If fldr.IsRootFolder = True Then
        Response.Write "This is the root folder." & ""<br>"
    Else
```
Response.Write "This folder isn't a root folder." & "

End If
' Create a new folder with the FileSystemObject object
fso.CreateFolder ("C:\Bogus")
Response.Write "Created folder C:\Bogus" & "<br>
' Print the base name of the folder.
Response.Write "Basename = " & fso.GetBaseName
' Delete the newly created folder.
fso.DeleteFolder ("C:\Bogus")
Response.Write "Deleted folder C:\Bogus" & "<br>
End Sub

This example shows how to use the **Folder** and **FileSystemObject** objects in JScript:

```javascript
function ShowFolderInfo()
{
    var fso, fldr, s = "";
    // Get instance of FileSystemObject.
    fso = new ActiveXObject("Scripting.FileSystemObject");
    // Get Drive object.
    fldr = fso.GetFolder("c:");
    // Print parent folder name.
    Response.Write("Parent folder name is: " + fldr + "<br>
    // Print drive name.
    Response.Write("Contained on drive " + fldr.Drive + "<br>
    // Print root file name.
    if (fldr.IsRootFolder)
Response.Write("This is the root folder.");
else
    Response.Write("This folder isn't a root folder.");
Response.Write("<br><br>"

// Create a new folder with the FileSystemObject obj
fso.CreateFolder ("C:\Bogus");
Response.Write("Created folder C:\Bogus" + "<br>"

// Print the base name of the folder.
Response.Write("Basename = " + fso.GetBaseName("

// Delete the newly created folder.
fso.DeleteFolder ("C:\Bogus");
Response.Write("Deleted folder C:\Bogus" + "<br>"
}


____________________________________________________________
There are two major categories of file manipulation:

- Creating, adding, or removing data, and reading files
- Moving, copying, and deleting files

Creating Files

There are three ways to create an empty text file (sometimes referred to as a "text stream").

The first way is to use the `CreateTextFile` method. The following example demonstrates how to create a text file using this method in VBScript:

```vbnet
Dim fso, f1
Set fso = CreateObject("Scripting.FileSystemObject")
Set f1 = fso.CreateTextFile("c:\testfile.txt", True)
```

To use this method in JScript, use this code:

```javascript
var fso, f1;
fso = new ActiveXObject("Scripting.FileSystemObject");
f1 = fso.CreateTextFile("c:\testfile.txt", true);
```

View this [sample code](#) to see how the `CreateTextFile` method is used in `FileSystemObject`.

The second way to create a text file is to use the `OpenTextFile` method of the `FileSystemObject` object with the `ForWriting` flag set. In VBScript,
the code looks like this example:

Dim fso, ts
Const ForWriting = 2
Set fso = CreateObject("Scripting. FileSystemObject")
Set ts = fso.OpenTextFile("c:\test.txt", ForWriting, True)

To create a text file using this method in JScript, use this code:

var fso, ts;
var ForWriting = 2;
fso = new ActiveXObject("Scripting.FileSystemObject");
ts = fso.OpenTextFile("c:\\test.txt", ForWriting, true);

A third way to create a text file is to use the OpenAsTextStream method with the ForWriting flag set. For this method, use the following code in VBScript:

Dim fso, f1, ts
Const ForWriting = 2
Set fso = CreateObject("Scripting.FileSystemObject")
fso.CreateTextFile("c:\test1.txt")
Set f1 = fso.GetFile("c:\test1.txt")
Set ts = f1.OpenAsTextStream(ForWriting, True)

In JScript, use the code in the following example:

var fso, f1, ts;
var ForWriting = 2;
fso = new ActiveXObject("Scripting.FileSystemObject")
fso.CreateTextFile("c:\test1.txt");
f1 = fso.GetFile("c:\test1.txt");
ts = f1.OpenAsTextStream(ForWriting, true);

Adding Data to the File

Once the text file is created, add data to the file using the following three steps:

1. Open the text file.
2. Write the data.
3. Close the file.

To open an existing file, use either the `OpenTextFile` method of the `FileSystemObject` object or the `OpenAsTextStream` method of the `File` object.

To write data to the open text file, use the `Write`, `WriteLine`, or `WriteBlankLines` methods of the `TextStream` object, according to the tasks outlined in the following table.

<table>
<thead>
<tr>
<th>Task</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write data to an open text file without a trailing newline character.</td>
<td>Write</td>
</tr>
<tr>
<td>Write data to an open text file with a trailing newline character.</td>
<td>WriteLine</td>
</tr>
<tr>
<td>Write one or more blank lines to an open text file.</td>
<td>WriteBlankLines</td>
</tr>
</tbody>
</table>

View this [sample code](#) to see how the `Write`, `WriteLine`, and `WriteBlankLines` methods are used in `FileSystemObject`. 
To close an open file, use the `Close` method of the `TextStream` object.

View this [sample code](#) to see how the `Close` method is used in `FileSystemObject`.

---

**Note**  The newline character contains a character or characters (depending on the operating system) to advance the cursor to the beginning of the next line (carriage return/line feed). Be aware that the end of some strings may already have such nonprinting characters.

---

The following VBScript example demonstrates how to open a file, use all three write methods to add data to the file, and then close the file:

```vbscript
Sub CreateFile()
    Dim fso, tf
    Set fso = CreateObject("Scripting.FileSystemObject"
    Set tf = fso.CreateTextFile("c:\testfile.txt", True)
    ' Write a line with a newline character.
    tf.WriteLine("Testing 1, 2, 3.")
    ' Write three newline characters to the file.
    tf.WriteBlankLines(3)
    ' Write a line.
    tf.Write("This is a test.")
    tf.Close
End Sub
```

This example demonstrates how to use the three methods in JScript:
function CreateFile()
{
    var fso, tf;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    tf = fso.CreateTextFile("c:\testfile.txt", true);
    // Write a line with a newline character.
    tf.WriteLine("Testing 1, 2, 3.");
    // Write three newline characters to the file.
    tf.WriteBlankLines(3);
    // Write a line.
    tf.Write("This is a test.");
    tf.Close();
}

**Reading Files**

To read data from a text file, use the **Read**, **ReadLine**, or **ReadAll** method of the **TextStream** object. The following table describes which method to use for various tasks.

<table>
<thead>
<tr>
<th>Task</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read a specified number of characters from a file.</td>
<td><strong>Read</strong></td>
</tr>
<tr>
<td>Read an entire line (up to, but not including, the newline character).</td>
<td><strong>ReadLine</strong></td>
</tr>
<tr>
<td>Read the entire contents of a text file.</td>
<td><strong>ReadAll</strong></td>
</tr>
</tbody>
</table>

View this [sample code](#) to see how the **ReadAll** and **ReadLine** methods are used in **FileSystemObject**.

If you use the **Read** or **ReadLine** method and want to skip to a particular
portion of data, use the **Skip** or **SkipLine** method. The resulting text of the read methods is stored in a string which can be displayed in a control, parsed by string functions (such as **Left**, **Right**, and **Mid**), concatenated, and so forth.

The following VBScript example demonstrates how to open a file, write to it, and then read from it:

Sub ReadFiles
    Dim fso, f1, ts, s
    Const ForReading = 1
    Set fso = CreateObject("Scripting.FileSystemObject'
    Set f1 = fso.CreateTextFile("c:\testfile.txt", True)
    ' Write a line.
    Response.Write "Writing file <br>"
    f1.WriteLine "Hello World"
    f1.WriteBlankLines(1)
    f1.Close
    ' Read the contents of the file.
    Response.Write "Reading file <br>"
    Set ts = fso.OpenTextFile("c:\testfile.txt", ForReading)
    s = ts.ReadLine
    Response.Write "File contents = " & s & ""
    ts.Close
End Sub

This code demonstrates the same thing in JScript:

```javascript
function ReadFiles()
{
```
var fso, f1, ts, s;
var ForReading = 1;
fso = new ActiveXObject("Scripting.FileSystemObject");
f1 = fso.CreateTextFile("c:\testfile.txt", true);
// Write a line.
Response.Write("Writing file <br>");
f1.WriteLine("Hello World");
f1.WriteBlankLines(1);
f1.Close();
// Read the contents of the file.
Response.Write("Reading file <br>");

Moving, Copying, and Deleting Files

The FSO object model has two methods each for moving, copying, and deleting files, as described in the following table.

<table>
<thead>
<tr>
<th>Task</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move a file</td>
<td>File.Move or FileSystemObject.MoveFile</td>
</tr>
<tr>
<td>Copy a file</td>
<td>File.Copy or FileSystemObject.CopyFile</td>
</tr>
<tr>
<td>Delete a file</td>
<td>File.Delete or FileSystemObject.DeleteFile</td>
</tr>
</tbody>
</table>

View this sample code to see two ways to delete a file in FileSystemObject.
The following VBScript example creates a text file in the root directory of drive C, writes some information to it, moves it to a directory called \tmp, makes a copy of it in a directory called \temp, then deletes the copies from both directories.

To run the following example, create directories named \tmp and \temp in the root directory of drive C:

```vbnet
Sub ManipFiles
    Dim fso, f1, f2, s
    Set fso = CreateObject("Scripting.FileSystemObject"
    Set f1 = fso.CreateTextFile("c:\testfile.txt", True)
    Response.Write "Writing file<br>"
    ' Write a line.
    f1.Write ("This is a test.")
    ' Close the file to writing.
    f1.Close
    Response.Write "Moving file to c:\tmp<br>"
    ' Get a handle to the file in root of C:\.
    Set f2 = fso.GetFile("c:\testfile.txt")
    ' Move the file to \tmp directory.
    f2.Move ("c:\tmp\testfile.txt")
    Response.Write "Copying file to c:\temp<br>"
    ' Copy the file to \temp.
    f2.Copy ("c:\temp\testfile.txt")
    Response.Write "Deleting files<br>"
    ' Get handles to files' current location.
    Set f2 = fso.GetFile("c:\tmp\testfile.txt")
    Set f3 = fso.GetFile("c:\temp\testfile.txt")
```
'Delete the files.
f2.Delete
f3.Delete
Response.Write "All done!"
End Sub

This code shows the same thing in JScript:

function ManipFiles()
{
    var fso, f1, f2, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
f1 = fso.CreateTextFile("c:\testfile.txt", true);
Response.Write("Writing file <br>");
    // Write a line.
    f1.Write("This is a test.");
    // Close the file to writing.
    f1.Close();
Response.Write("Moving file to c:\tmp <br>");
    // Get a handle to the file in root of C:.
    f2 = fso.GetFile("c:\testfile.txt");
    // Move the file to \tmp directory.
    f2.Move ("c:\tmp\testfile.txt");
Response.Write("Copying file to c:\temp <br>");
    // Copy the file to \temp.
    f2.Copy ("c:\temp\testfile.txt");
Response.Write("Deleting files <br>");
    // Get handles to files' current location.
f2 = fso.GetFile("c:\tmp\testfile.txt");
f3 = fso.GetFile("c:\temp\testfile.txt");
// Delete the files.
f2.Delete();
f3.Delete();
Response.Write("All done!");
}
The sample code described in this section provides a real-world example that demonstrates many of the features available in the FileSystemObject object model. This code shows how all the features of the object model work together, and how to use those features effectively in your own code.

Note that since this code is fairly generic, some additional code and a little tweaking are needed to make this code actually run on your machine. These changes are necessary because of the different ways input and output to the user is handled between Active Server Pages and the Windows Scripting Host.

To run this code on an Active Server Page, use the following steps:


2. Copy the following sample code into that file between the <BODY;>...</BODY> tags.

3. Enclose all the code within <%...%> tags.

4. Move the Option Explicit statement from its current position in the code to the very top of your HTML page, positioning it even before the opening <HTML> tag.

5. Place <%...%> tags around the Option Explicit statement to ensure that it's run on the server side.

6. Add the following code to the end of the sample code:

   Sub Print(x)
Response.Write "<PRE><FONT; FACE=""Courier New"" SIZE=""1"">"
Response.Write x
Response.Write "</FONT></PRE>"
End Sub
Main

The previous code adds a print procedure that will run on the server side, but display results on the client side. To run this code on the Windows Scripting Host, add the following code to the end of the sample code:

Sub Print(x)
    WScript.Echo x
End Sub
Main

The code is contained in the following section:

........................................................................

' FileSystemObject Sample Code
'
' Copyright 1998 Microsoft Corporation. All Rights Reserved.
',
........................................................................

Option Explicit

........................................................................

',

' Regarding code quality:
',

' 1) The following code does a lot of string manipulation by concatenating short strings together with the "&" operator. Since string concatenation is expensive, this is a very inefficient way to write code. However, it is a very maintainable way to write code, and is used here because this program perf...
' disk operations, and because the disk is much slower than the memory oper  
' concatenate the strings. Keep in mind that this is demonstration code, not pi 
'
' 2) "Option Explicit" is used, because declared variable access is slightly faste  
' undeclared variable access. It also prevents bugs from creeping into your cc  
' when you misspell DriveTypeCDROM as DriveTypeCDORM.
'
' 3) Error handling is absent from this code, to make the code more readable. A  
' precautions have been taken to ensure that the code will not error in common  
' systems can be unpredictable. In production code, use On Error Resume Ne  
' Err object to trap possible errors.

' Some handy global variables
'
Dim TabStop
Dim NewLine

Const TestDrive = "C"
Const TestFilePath = "C:\Test"

Const DriveTypeRemovable = 1
Const DriveTypeFixed = 2
Const DriveTypeNetwork = 3
Const DriveTypeCDROM = 4
Const DriveTypeRAMDisk = 5
' Constants returned by File.Attributes
'
Const FileAttrNormal = 0
Const FileAttrReadOnly = 1
Const FileAttrHidden = 2
Const FileAttrSystem = 4
Const FileAttrVolume = 8
Const FileAttrDirectory = 16
Const FileAttrArchive = 32
Const FileAttrAlias = 64
Const FileAttrCompressed = 128

' Constants for opening files
'
Const OpenFileForReading = 1
Const OpenFileForWriting = 2
Const OpenFileForAppending = 8

' ShowDriveType
'
' Purpose:
'
' Generates a string describing the drive type of a given Drive object.
'
' Demonstrates the following
Function ShowDriveType(Drive)

    Dim S

    Select Case Drive.DriveType
    Case DriveTypeRemovable
        S = "Removable"
    Case DriveTypeFixed
        S = "Fixed"
    Case DriveTypeNetwork
        S = "Network"
    Case DriveTypeCDROM
        S = "CD-ROM"
    Case DriveTypeRAMDisk
        S = "RAM Disk"
    Case Else
        S = "Unknown"
    End Select

    ShowDriveType = S

End Function

' - Drive.DriveType
'

' ShowFileAttr
'
' Purpose:
'
' Generates a string describing the attributes of a file or folder.
'
' Demonstrates the following
Function ShowFileAttr(File) ' File can be a file or folder

    Dim S
    Dim Attr

    Attr = File.Attributes

    If Attr = 0 Then
        ShowFileAttr = "Normal"
        Exit Function
    End If

    If Attr And FileAttrDirectory Then S = S & "Directory 
    If Attr And FileAttrReadOnly Then S = S & "Read-Only 
    If Attr And FileAttrHidden Then S = S & "Hidden 
    If Attr And FileAttrSystem Then S = S & "System 
    If Attr And FileAttrVolume Then S = S & "Volume 
    If Attr And FileAttrArchive Then S = S & "Archive 
    If Attr And FileAttrAlias Then S = S & "Alias 
    If Attr And FileAttrCompressed Then S = S & "Compressed 

    ShowFileAttr = S

End Function
Generates a string describing the current state of the available drives.

Demonstrates the following

- FileSystemObject.Drives
- Iterating the Drives collection
- Drives.Count
- Drive.AvailableSpace
- Drive.DriveLetter
- Drive.DriveType
- Drive.FileSystem
- Drive.FreeSpace
- Drive.IsReady
- Drive.Path
- Drive.SerialNumber
- Drive.ShareName
- Drive.TotalSize
- Drive.VolumeName

Function GenerateDriveInformation(FSO)

    Dim Drives
    Dim Drive
    Dim S

    Set Drives = FSO.Drives

    S = "Number of drives:" & TabStop & Drives.Count & NewLine & NewLine

    ' Construct 1st line of report.
    S = S & String(2, TabStop) & "Drive"
    S = S & String(3, TabStop) & "File"
    S = S & TabStop & "Total"
    S = S & TabStop & "Free"
S = S & TabStop & "Available"
S = S & TabStop & "Serial" & NewLine

' Construct 2nd line of report.
S = S & "Letter"
S = S & TabStop & "Path"
S = S & TabStop & "Type"
S = S & TabStop & "Ready?"
S = S & TabStop & "Name"
S = S & TabStop & "System"
S = S & TabStop & "Space"
S = S & TabStop & "Space"
S = S & TabStop & "Space"
S = S & TabStop & "Number" & NewLine

' Separator line.
S = S & String(105, "-") & NewLine

For Each Drive In Drives

    S = S & Drive.DriveLetter
    S = S & TabStop & Drive.Path
    S = S & TabStop & ShowDriveType(Drive)
    S = S & TabStop & Drive.IsReady

    If Drive.IsReady Then
        If DriveTypeNetwork = Drive.DriveType Then
            S = S & TabStop & Drive.ShareName
        Else
            S = S & TabStop & Drive.VolumeName
        End If
    End If

    S = S & TabStop & Drive.FileSystem
    S = S & TabStop & Drive.TotalSize
    S = S & TabStop & Drive.FreeSpace
    S = S & TabStop & Drive.AvailableSpace
    S = S & TabStop & Hex(Drive.SerialNumber)
End If

S = S & NewLine

Next

GenerateDriveInformation = S

End Function

' GenerateFileInformation
'
' Purpose:
'
' Generates a string describing the current state of a file.
'
' Demonstrates the following
'
' - File.Path
' - File.Name
' - File.Type
' - File.DateCreated
' - File.DateLastAccessed
' - File.DateLastModified
' - File.Size
'
Function GenerateFileInformation(File)

    Dim S

    S = NewLine & "Path:" & TabStop & File.Path
    S = S & NewLine & "Name:" & TabStop & File.Name
S = S & NewLine & "Type:" & TabStop & File.Type
S = S & NewLine & "Attribs:" & TabStop & ShowFileAttr(File)
S = S & NewLine & "Created:" & TabStop & File.DateCreated
S = S & NewLine & "Modified:" & TabStop & File.DateLastModified

GenerateFileInformation = S

End Function

'''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
End Function

'''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
Set Files = Folder.Files

If 1 = Files.Count Then
    S = S & "There is 1 file" & NewLine
Else
    S = S & "There are " & Files.Count & " files" & NewLine
End If

If Files.Count <> 0 Then
    For Each File In Files
        S = S & GenerateFileInformation(File)
    Next
End If

Set SubFolders = Folder.SubFolders

If 1 = SubFolders.Count Then
    S = S & NewLine & "There is 1 sub folder" & NewLine & NewLine
Else
    S = S & NewLine & "There are " & SubFolders.Count & " sub folders"
End If

If SubFolders.Count <> 0 Then
    For Each SubFolder In SubFolders
        S = S & GenerateFolderInformation(SubFolder)
    Next
    S = S & NewLine
    For Each SubFolder In SubFolders
        S = S & GenerateAllFolderInformation(SubFolder)
    Next
End If
GenerateAllFolderInformation = S

End Function

''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''

' GenerateTestInformation
'
' Purpose:
'
' Generates a string describing the current state of the C:\Test
' folder and all files and subfolders.
'
' Demonstrates the following
'
' - FileSystemObject.DriveExists
' - FileSystemObject.FolderExists
' - FileSystemObject.GetFolder
'
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''

Function GenerateTestInformation(FSO)

    Dim TestFolder
    Dim S

    If Not FSO.DriveExists(TestDrive) Then Exit Function
    If Not FSO.FolderExists(TestFilePath) Then Exit Function

    Set TestFolder = FSO.GetFolder(TestFilePath)

    GenerateTestInformation = GenerateAllFolderInformation(TestFolder)

End Function
Sub DeleteTestDirectory(FSO)
    Dim TestFolder
    Dim SubFolder
    Dim File
    
    ' Two ways to delete a file:
    FSO.DeleteFile(TestFilePath & "\Beatles\OctopusGarden.txt")
    Set File = FSO.GetFile(TestFilePath & "\Beatles\BathroomWindow.txt")
    File.Delete
    
    ' Two ways to delete a folder:
    FSO.DeleteFolder(TestFilePath & "\Beatles")
FSO.DeleteFile(TestFilePath & "\ReadMe.txt")

Set TestFolder = FSO.GetFolder(TestFilePath)
TestFolder.Delete

End Sub

' CreateLyrics

' Purpose:

' Builds a couple of text files in a folder.

' Demonstrates the following

' - FileSystemObject.CreateTextFile
' - TextStream.WriteLine
' - TextStream.Write
' - TextStream.WriteBlankLines
' - TextStream.Close

Sub CreateLyrics(Folder)

Dim TextStream

Set TextStream = Folder.CreateTextFile("OctopusGarden.txt")

TextStream.Write("Octopus' Garden ") ' Note that this does not add a line
TextStream.WriteLine("(by Ringo Starr)")
TextStream.WriteBlankLines(1)
TextStream.WriteLine("I'd like to be under the sea in an octopus' garden i
TextStream.WriteLine("He'd let us in, knows where we've been -- in his o
TextStream.WriteLine("She Came In Through The Bathroom Window (by Lennon/McCartney)")
TextStream.WriteLine("She came in through the bathroom window protected by a silver spoon")
TextStream.WriteLine("But now she sucks her thumb and wanders by the town lagoon")

End Sub

'''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
Dim TextStream
Dim S
Dim File

' There are several ways to open a text file, and several ways to read the
' data out of a file. Here's two ways to do each:

Set TextStream = FSO.OpenTextFile(TestFilePath & "\Beatles\OctopusGarden.txt", OpenFileForReading)
S = TextStream.ReadAll & NewLine & NewLine
TextStream.Close

Set File = FSO.GetFile(TestFilePath & "\Beatles\BathroomWindow.txt")
Set TextStream = File.OpenAsTextStream(OpenFileForReading)
Do While Not TextStream.AtEndOfStream
    S = S & TextStream.ReadLine & NewLine
Loop
TextStream.Close

GetLyrics = S

End Function

______________________________________________________________

' BuildTestDirectory

' Purpose:

' Builds a directory hierarchy to demonstrate the FileSystemObject.

' We'll build a hierarchy in this order:

' C:\Test
' C:\Test\ReadMe.txt
' C:\Test\Beatles
Demonstrates the following

- FileSystemObject.DriveExists
- FileSystemObject.FolderExists
- FileSystemObject.CreateFolder
- FileSystemObject.CreateTextFile
- Folders.Add
- Folder.CreateTextFile
- TextStream.WriteLine
- TextStream.Close

Function BuildTestDirectory(FSO)

    Dim TestFolder
    Dim SubFolders
    Dim SubFolder
    Dim TextStream

    ' Bail out if (a) the drive does not exist, or if (b) the directory being built
    ' already exists.

    If Not FSO.DriveExists(TestDrive) Then
        BuildTestDirectory = False
        Exit Function
    End If

    If FSO.FolderExists(TestFilePath) Then
        BuildTestDirectory = False
        Exit Function
    End If
Set TestFolder = FSO.CreateFolder(TestFilePath)

Set TextStream = FSO.CreateTextFile(TestFilePath & "\ReadMe.txt")
TextStream.WriteLine("My song lyrics collection")
TextStream.Close

Set SubFolders = TestFolder.SubFolders

Set SubFolder = SubFolders.Add("Beatles")
CreateLyrics SubFolder

BuildTestDirectory = True

End Function

''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
'
The main routine
'
'''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
If Not BuildTestDirectory(FSO) Then
    Print "Test directory already exists or cannot be created. Cannot con"
    Exit Sub
End If

Print GenerateDriveInformation(FSO) & NewLine & NewLine

Print GenerateTestInformation(FSO) & NewLine & NewLine

Print GetLyrics(FSO) & NewLine & NewLine

DeleteTestDirectory(FSO)

End Sub
<table>
<thead>
<tr>
<th>Category</th>
<th>Feature/Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array Handling</td>
<td>Array join, length, reverse, sort</td>
</tr>
<tr>
<td>Assignments</td>
<td>Assign (=) Compound Assign (OP=)</td>
</tr>
<tr>
<td>Booleans</td>
<td>Boolean</td>
</tr>
<tr>
<td>Comments</td>
<td>/* ... */ or //</td>
</tr>
<tr>
<td>Constants/Literals</td>
<td>NaN null true, false Infinity undefined</td>
</tr>
<tr>
<td>Control flow</td>
<td>break continue for for...in if...else return while</td>
</tr>
<tr>
<td>Dates and Time</td>
<td>Date getDate, getDay, getFullYear, getHours, getMilliseconds, getMinutes, getMonth, getSeconds, getTime, getTimezoneOffset, getYear, getUTCDate, getUTCDay, getUTCFullYear, getUTCHours, getUTCMilliseconds, getUTCMilliseconds, getUTCMinutes, getUTCMonth, getUTCSecs, setDate, setFullYear, setHours, setMilliseconds, setMinutes, setMonth, setSeconds, setTime, setYear, setUTCDate, setUTCFullYear, setUTCHours,</td>
</tr>
<tr>
<td>Declarations</td>
<td>function, new, this, var, with</td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>Function Creation</td>
<td>Function, arguments, length</td>
</tr>
<tr>
<td>Global Methods</td>
<td>Global escape, unescape, eval, isFinite, isNaN, parseInt, parseFloat</td>
</tr>
<tr>
<td>Math</td>
<td>Math abs, acos, asin, atan, atan2, ceil, cos, exp, floor, log, max, min, pow, random, round, sin, sqrt, tan, E, LN2, LN10, LOG2E, LOG10E, PI, SQRT1_2, SQRT2</td>
</tr>
<tr>
<td>Numbers</td>
<td>Number MAX_VALUE, MIN_VALUE, NaN, NEGATIVE_INFINITY, POSITIVE_INFINITY</td>
</tr>
<tr>
<td>Object Creation</td>
<td>Object new constructor, prototype, toString, valueOf</td>
</tr>
<tr>
<td>Operators</td>
<td>Addition (+), Subtraction (-), Modulus arithmetic (%), Multiplication (*), Division (/), Negation (-), Equality (==), Inequality (!=), Less Than (&lt;), Less Than or Equal To (&lt;=), Greater Than (&gt;) Greater Than or Equal To (&gt;=), Logical And(&amp;&amp;), Or (!</td>
</tr>
<tr>
<td>Conditional (?)</td>
<td>Comma (,)</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------</td>
</tr>
<tr>
<td><strong>Objects</strong></td>
<td><strong>Array</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Function</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Number</strong></td>
</tr>
<tr>
<td><strong>Strings</strong></td>
<td><strong>String</strong></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Feature/Keyword</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>Array Handling</td>
<td><code>concat</code>, <code>slice</code></td>
</tr>
<tr>
<td></td>
<td><code>VBAArray</code></td>
</tr>
<tr>
<td></td>
<td><code>dimensions</code>, <code>getItem</code>, <code>lbound</code>, <code>toArray</code>, <code>ubound</code></td>
</tr>
<tr>
<td>Conditional Compilation</td>
<td><code>@cc_on</code></td>
</tr>
<tr>
<td></td>
<td><code>@if Statement</code></td>
</tr>
<tr>
<td></td>
<td><code>@set Statement</code></td>
</tr>
<tr>
<td></td>
<td><code>Conditional Compilation Variables</code></td>
</tr>
<tr>
<td>Control flow</td>
<td><code>do...while</code></td>
</tr>
<tr>
<td></td>
<td><code>Labeled</code></td>
</tr>
<tr>
<td></td>
<td><code>switch</code></td>
</tr>
<tr>
<td>Dates and Time</td>
<td><code>getVarDate</code></td>
</tr>
<tr>
<td>Enumeration</td>
<td><code>Enumerator</code></td>
</tr>
<tr>
<td></td>
<td><code>atEnd</code>, <code>item</code>, <code>moveFirst</code>, <code>moveNext</code></td>
</tr>
<tr>
<td>Error Handling</td>
<td><code>Error</code></td>
</tr>
<tr>
<td></td>
<td><code>description</code>, <code>number</code></td>
</tr>
<tr>
<td></td>
<td><code>throw</code>, <code>try...catch</code></td>
</tr>
<tr>
<td>Function Creation</td>
<td><code>caller</code></td>
</tr>
<tr>
<td>Operators</td>
<td><code>Identity (===)</code>, <code>Nonidentity (!==)</code></td>
</tr>
<tr>
<td>Objects</td>
<td><code>Enumerator</code></td>
</tr>
<tr>
<td></td>
<td><code>RegExp</code></td>
</tr>
<tr>
<td></td>
<td><code>Regular Expression</code></td>
</tr>
<tr>
<td></td>
<td><code>VBAArray</code></td>
</tr>
<tr>
<td></td>
<td><code>ActiveXObject</code></td>
</tr>
<tr>
<td></td>
<td><code>GetObject</code></td>
</tr>
<tr>
<td>Regular</td>
<td><code>RegExp</code></td>
</tr>
<tr>
<td>Expressions and Pattern Matching</td>
<td>index, input, lastIndex, $1...$9, source, compile, exec, test</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>Regular Expression Syntax</td>
</tr>
<tr>
<td>Script Engine Identification</td>
<td>ScriptEngine</td>
</tr>
<tr>
<td></td>
<td>ScriptEngineBuildVersion</td>
</tr>
<tr>
<td></td>
<td>ScriptEngineMajorVersion</td>
</tr>
<tr>
<td></td>
<td>ScriptEngineMinorVersion</td>
</tr>
<tr>
<td>Strings</td>
<td>concat, slice</td>
</tr>
<tr>
<td></td>
<td>match, replace, search</td>
</tr>
<tr>
<td></td>
<td>anchor, big, blink, bold, fixed, fontcolor, fontsize, italics, link, small, strike, sub, sup</td>
</tr>
</tbody>
</table>
## Scripting Run-Time Features

<table>
<thead>
<tr>
<th>Category</th>
<th>Feature/Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collections</td>
<td>Drives, Files, Folders</td>
</tr>
<tr>
<td>Data Storage</td>
<td>Dictionary</td>
</tr>
<tr>
<td>Dictionary</td>
<td>Add, Exists, Items, Keys, Remove, RemoveAll, Count, Item, Key</td>
</tr>
<tr>
<td>File System</td>
<td>Drive, File, FileSystemObject, Folder, TextStream</td>
</tr>
<tr>
<td>FileSystemObject</td>
<td>BuildPath, CopyFile, CopyFolder, CreateFolder, CreateTextFile, DeleteFile, DeleteFolder, DriveExists, FileExists, FolderExists, GetAbsolutePathName, GetBaseName, GetDrive, GetDriveName, GetFile, GetExtensionName, GetFileName, GetFolder, GetParentFolderName, GetSpecialFolder, GetTempName, MoveFile, MoveFolder, OpenTextFile</td>
</tr>
<tr>
<td>Drive, Drives</td>
<td>Drives</td>
</tr>
<tr>
<td>--------------</td>
<td>--------</td>
</tr>
<tr>
<td>File, Files</td>
<td>Add</td>
</tr>
<tr>
<td>Folder, Folders</td>
<td>Close</td>
</tr>
</tbody>
</table>
Returns the absolute value of a number.

Syntax

Math.abs(number)

The number argument is a numeric expression for which the absolute value is sought.

Remarks

The return value is the absolute value of the number argument.

The following example illustrates the use of the abs method:

```javascript
function ComparePosNegVal(n)
{
    var s;
    var v1 = Math.abs(n);
    var v2 = Math.abs(-n);
    if (v1 == v2)
        s = "The absolute values of " + n
```
s += -n + " are identical."
return(s);
}
acos Method

See Also

Applies To

Description

Returns the arccosine of a number.

Syntax

Math.acos(number)

The number argument is a numeric expression for which the arccosine is sought.

Remarks

The return value is the arccosine of the number argument.
ActiveXObject Object

See Also

Description

Enables and returns a reference to an Automation object.

Syntax

```javascript
var newObject = new ActiveXObject("servername.typename", "location")
```

The `ActiveXObject` object syntax and has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>servername</code></td>
<td>Required. The name of the application providing the object.</td>
</tr>
<tr>
<td><code>typename</code></td>
<td>Required. The type or class of the object to create.</td>
</tr>
<tr>
<td><code>location</code></td>
<td>Optional. The name of the network server where the object is to be created.</td>
</tr>
</tbody>
</table>

Remarks

Automation servers provide at least one type of object. For example, a word-processing application may provide an application object, a document object, and a toolbar object.

To create an Automation object, assign the new `ActiveXObject` to an object variable:
This code starts the application creating the object (in this case, a Microsoft Excel worksheet). Once an object is created, you refer to it in code using the object variable you defined. In the following example, you 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. For example:

```
// Make Excel visible through the Application object.
ExcelSheet.Application.Visible = true;
// Place some text in the first cell of the sheet.
ExcelSheet.ActiveSheet.Cells(1,1).Value = "This is column A, row 1";
// Save the sheet.
ExcelSheet.SaveAs("C:\TEST.XLS");
// Close Excel with the Quit method on the Application object.
ExcelSheet.Application.Quit();
// Release the object variable.
ExcelSheet = "";
```

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 `ActiveXObject`. That name is the same as the machine name portion of a sharename. For a network
share named "\\myserver\public", the servername is "myserver". In addition, you can specify servername using DNS format or an IP address.

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

```vbnet
Function GetVersion {
    var XLApp = CreateObject("Excel.Application", "MyServer");
    return(XLApp.Version);
}
```

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

Used to sum two numbers or perform string concatenation.

Syntax

\[ result = expression1 + expression2 \]

The + operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( result )</td>
<td>Any variable.</td>
</tr>
<tr>
<td>( expression1 )</td>
<td>Any expression.</td>
</tr>
<tr>
<td>( expression2 )</td>
<td>Any expression.</td>
</tr>
</tbody>
</table>

Remarks

The underlying subtype of the expressions determines the behavior of the + operator.

<table>
<thead>
<tr>
<th>If</th>
<th>Then</th>
</tr>
</thead>
<tbody>
<tr>
<td>Both expressions are numeric or Boolean</td>
<td>Add.</td>
</tr>
<tr>
<td>Both expressions are strings</td>
<td>Concatenate.</td>
</tr>
<tr>
<td>One expression is numeric and</td>
<td></td>
</tr>
<tr>
<td>the other is a string</td>
<td>Concatenate.</td>
</tr>
</tbody>
</table>

For information on when a run-time error is generated by the + operator, see the Operator Behavior table.
anchor Method

See Also

Applies To

Description

Places an HTML anchor with a NAME attribute around specified text in the object.

Syntax

strVariable.anchor(anchorstring)
"String Literal".anchor(anchorstring)

The anchorstring argument is text you want to place in the NAME attribute of an HTML anchor.

Remarks

Call the anchor method to create a named anchor out of a String object. The following example demonstrates how the anchor method accomplishes this:

var strVariable = "This is an anchor" ;
strVariable = strVariable.anchor("Anchor1");

The value of strVariable after the last statement is:

<A NAME="Anchor1">This is an anchor</A>

No checking is done to see if the tag has already been applied to the string.
Array Object

Description

Provides support for creation of arrays of any data type.

Syntax

```
new Array()
new Array(size)
new Array(element0, element1, ..., elementn)
```

The `Array` object creation syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>size</code></td>
<td>The size of the array. As arrays are zero-based, created elements will have indexes from zero to <code>size -1</code>.</td>
</tr>
<tr>
<td><code>element0,...,elementn</code></td>
<td>The elements to place in the array. This creates an array with ( n + 1 ) elements, and a <strong>length</strong> of ( n ).</td>
</tr>
</tbody>
</table>

Remarks
After an array is created, the individual elements of the array can be accessed using [ ] notation, for example:

```javascript
var my_array = new Array();
for (i = 0; i < 10; i++)
{
    my_array[i] = i;
}
x = my_array[4];
```

Since arrays in Microsoft JScript are zero-based, the last statement in the preceding example accesses the fifth element of the array. That element contains the value 4.

If only one argument is passed to the `Array` constructor, and the argument is a number, it is coerced into an unsigned integer, and the value is used as the size of the array. Otherwise, the parameter passed in is used as the only element of the array.
Description

Returns the arcsine of a number.

Syntax

Math.asin(number)

The number argument is a numeric expression for which the arcsine is sought.

Remarks

The return value is the arcsine of its numeric argument.
Description

Assigns a value to a variable.

Syntax

\[ result = expression \]

The = operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any variable.</td>
</tr>
<tr>
<td>expression</td>
<td>Any numeric expression.</td>
</tr>
</tbody>
</table>

Remarks

As the = operator behaves like other operators, expressions using it have a value in addition to assigning that value into variable. This means that you can chain assignment operators as follows:

\[ j = k = l = 0; \]

\[ j, k, \text{ and } l \] equal zero after the example statement is executed.
### Description

Returns the arctangent of a number.

### Syntax

```javascript
Math.atan(number)
```

The `number` argument is a numeric expression for which the arctangent is sought.

### Remarks

The return value is the arctangent of its numeric argument.
atan2 Method

Description

Returns the angle (in radians) from the X axis to a point (y,x).

Syntax

Math.atan2(y, x)

The `atan2` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math</td>
<td>Required. Invokes the intrinsic <code>Math</code> object.</td>
</tr>
<tr>
<td>x</td>
<td>Required. A numeric expression representing the cartesian x-coordinate.</td>
</tr>
<tr>
<td>y</td>
<td>Required. A numeric expression representing the cartesian y-coordinate.</td>
</tr>
</tbody>
</table>

Remarks

The return value is between -pi and pi, representing the angle of the supplied (y,x) point.
atEnd Method

Description

Returns a Boolean value indicating if the enumerator is at the end of the collection.

Syntax

```
myEnum.atEnd()
```

The `myEnum` argument is any `Enumerator` object.

Return Value

The `atEnd` method returns `true` if the current item is the last one in the collection, the collection is empty, or the current item is `undefined`. Otherwise, it returns `false`.

Remarks

In following code, the `atEnd` method is used to determine if the end of a list of drives has been reached:

```javascript
function ShowDriveList()
{
    var fso, s, n, e, x;
```
fso = new ActiveXObject("Scripting.FileSystemObject");
e = new Enumerator(fso.Drives);
s = "";
for (; !e.atEnd(); e.moveNext())
{
    x = e.item();
    s = s + x.DriveLetter;
    s += " - ";
    if (x.DriveType == 3)
        n = x.ShareName;
    else if (x.IsReady)
        n = x.VolumeName;
    else
        n = "[Drive not ready]";
    s += n + "<br>";
}
return(s);
}
**Description**

Places HTML `<BIG>` tags around text in a `String` object.

**Syntax**

```
strVariable.big()
"String Literal".big()
```

**Remarks**

The example that follows shows how the `big` method works:

```javascript
var strVariable = "This is a string object";
strVariable = strVariable.big();
```

The value of `strVariable` after the last statement is:

```
<BIG>This is a string object</BIG>
```

No checking is done to see if the tag has already been applied to the string.
Description

Performs a bitwise AND on two expressions.

Syntax

result = expression1 & expression2

The & operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any variable.</td>
</tr>
<tr>
<td>expression1</td>
<td>Any expression.</td>
</tr>
<tr>
<td>expression2</td>
<td>Any expression.</td>
</tr>
</tbody>
</table>

Remarks

The & operator looks at the binary representation of the values of two expressions and does a bitwise AND operation on them. The result of this operation behaves as follows:

```
 0101  (expression1)
 1100  (expression2)
----
 0100  (result)
```
Any time both of the expressions have a 1 in a digit, the result has a 1 in that digit. Otherwise, the result has a 0 in that digit.

For information on when a run-time error is generated by the & operator, see the Operator Behavior table.
See Also

Description

Shifts the bits of an expression to the left.

Syntax

\[ result = expression1 \ll expression2 \]

The `\ll` operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>result</code></td>
<td>Any variable.</td>
</tr>
<tr>
<td><code>expression1</code></td>
<td>Any expression.</td>
</tr>
<tr>
<td><code>expression2</code></td>
<td>Any expression.</td>
</tr>
</tbody>
</table>

Remarks

The `\ll` operator shifts the bits of `expression1` left by the number of bits specified in `expression2`. For example:

```javascript
var temp
temp = 14 \ll 2
```

The variable `temp` has a value of 56 because 14 (00001110 in binary) shifted left two bits equals 56 (00111000 in binary).

For information on when a run-time error is generated by the `\ll` operator, see the
Operator Behavior table.
Description

Performs a bitwise NOT (negation) on an expression.

Syntax

result = ~ expression

The ~ operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any variable.</td>
</tr>
<tr>
<td>expression</td>
<td>Any expression.</td>
</tr>
</tbody>
</table>

Remarks

All unary operators, such as the ~ operator, evaluate expressions as follows:

- If applied to undefined or null expressions, a run-time error is raised.
- Objects are converted to strings.
- Strings are converted to numbers if possible. If not, a run-time error is raised.
- Boolean values are treated as numbers (0 if false, 1 if true).
The operator is applied to the resulting number.

The ~ operator looks at the binary representation of the values of the expression and does a bitwise negation operation on it. The result of this operation behaves as follows:

```
0101   (expression)
----    
1010   (result)
```

Any digit that is a 1 in the expression becomes a 0 in the result. Any digit that is a 0 in the expression becomes a 1 in the result.
Description

Performs a bitwise OR on two expressions.

Syntax

\[ result = expression1 \ | \ expression2 \]

The | operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{result}</td>
<td>Any \textit{variable}.</td>
</tr>
<tr>
<td>\textit{expression1}</td>
<td>Any \textit{expression}.</td>
</tr>
<tr>
<td>\textit{expression2}</td>
<td>Any expression.</td>
</tr>
</tbody>
</table>

Remarks

The | operator looks at the binary representation of the values of two expressions and does a bitwise OR operation on them. The result of this operation behaves as follows:

\[
\begin{array}{c}
0101 \quad \text{(expression1)} \\
1100 \quad \text{(expression2)} \\
---- \\
1101 \quad \text{(result)}
\end{array}
\]
Any time either of the expressions has a 1 in a digit, the result will have a 1 in that digit. Otherwise, the result will have a 0 in that digit.

For information on when a run-time error is generated by the | operator, see the Operator Behavior table.
See Also

Description

Shifts the bits of an expression to the right, maintaining sign.

Syntax

\[ result = expression1 \gg expression2 \]

The \( \gg \) operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any variable</td>
</tr>
<tr>
<td>expression1</td>
<td>Any expression</td>
</tr>
<tr>
<td>expression2</td>
<td>Any expression</td>
</tr>
</tbody>
</table>

Remarks

The \( \gg \) operator shifts the bits of \( expression1 \) right by the number of bits specified in \( expression2 \). The sign bit of \( expression1 \) is used to fill the digits from the left. Digits shifted off the right are discarded. For example, after the following code is evaluated, \( temp \) has a value of -4: 14 (11110010 in binary) shifted right two bits equals -4 (1111100 in binary).

\[
\text{var temp}
\text{temp} = -14 \gg 2
\]
For information on when a run-time error is generated by the >> operator, see the Operator Behavior table.
**Description**

Places HTML `<BLINK>` tags around text in a `String` object.

**Syntax**

```
strVariable.blink();
"String Literal".blink();
```

**Remarks**

The following example demonstrates how the `blink` method works:

```javascript
var strVariable = "This is a string object";
strVariable = strVariable.blink();
```

The value of `strVariable` after the last statement is:

```
<BLINK>This is a string object</BLINK>
```

No checking is done to see if the tag has already been applied to the string.

The `<BLINK>` tag is not supported in Microsoft Internet Explorer.
**bold Method**

**See Also**

**Applies To**

---

**Description**

Places HTML `<B>` tags around text in a `String` object.

**Syntax**

```
strVariable.bold()
"String Literal".bold()
```

**Remarks**

The following example demonstrates how the `bold` method works:

```javascript
var strVariable = "This is a string object";
strVariable = strVariable.bold();
```

The value of `strVariable` after the last statement is:

```
<B>This is a string object</B>
```

No checking is done to see if the tag has already been applied to the string.
**Boolean Object**

**See Also**

**Methods**

**Properties**

---

**Description**

Creates a new Boolean value.

**Syntax**

```javascript
var variablename = new Boolean(boolvalue)
```

The optional `boolvalue` argument is the initial Boolean value for the new object. If this value is omitted, or is `false`, `0`, `null`, `NaN`, or an empty string, the initial value of the Boolean object is `false`. Otherwise, the initial value is `true`.

**Remarks**

The Boolean object is a wrapper for the Boolean data type. JScript implicitly uses the Boolean object whenever a Boolean data type is converted to a Boolean object.

You rarely call the Boolean object explicitly.
Description

Terminates the current loop, or if in conjunction with a label, terminates the associated statement.

Syntax

```
break [label];
```

The optional label argument specifies the label of the statement you are breaking from.

Remarks

You typically use the break statement in switch statements and while, for, for...in, or do...while loops. You most commonly use the label argument in switch statements, but it can be used in any statement, whether simple or compound.

Executing the break statement exits from the current loop or statement, and begins script execution with the statement immediately following.

The following example illustrates the use of the break statement:

```javascript
function BreakTest(breakpoint)
{
    var i = 0;
}
while (i < 100) {
    if (i == breakpoint) {
        break;
        i++;
    }
    return(i);
}
try...catch Statement

See Also

Description

Implements error handling for JScript.

Syntax

```
try
  tryStatement
catch(exception)
  catchStatement
```

The `try...catch` statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tryStatement</td>
<td>Statement where an error can occur. Can be a <code>compound statement</code>.</td>
</tr>
<tr>
<td>exception</td>
<td>Any <code>variable</code> name. The initial value of <code>exception</code> is the value of the thrown error.</td>
</tr>
<tr>
<td>catchStatement</td>
<td>Statement to handle errors occurring in the associated <code>tryStatement</code>. Can be a <code>compound statement</code>.</td>
</tr>
</tbody>
</table>
Remarks

The **try...catch** statement provides a way to handle some or all of the possible errors that may occur in a given block of code, while still running code. If errors occur that the programmer has not handled, JScript simply provides its normal error message to a user, as if there was no error handling.

The *tryStatement* argument contains code where an error can occur, while *catchStatement* contains the code to handle any error that does occur. If an error occurs in the *tryStatement*, program control is passed to *catchStatement* for disposition. The initial value of *exception* is the value of the error that occurred in *tryStatement*.

If the error cannot be handled in the *catchStatement* associated with the *tryStatement* where the error occurred, use the *throw* statement to propagate, or *rethrow*, the error to a higher-level error handler.

The following example throws an error based on a passed-in value. It then illustrates how that error is handled in a hierarchy of **try...catch** statements:

```javascript
function TryCatchDemo(x)
{
    try {
        try {
            if (x == 0) {  // Evaluate argument.
                throw "x equals zero";
            } else
                throw "x does not equal zero";
        }
    }
}
```
```javascript
catch(e) {
    if (e == "x equals zero")
        return(e + " handled locally.");
    else
        throw e;
}

try {
    document.write(TryCatchDemo(0));
    document.write(TryCatchDemo(1));
}
```

@cc_on Statement

See Also

Description

Activates conditional compilation support.

Syntax

@cc_on

Remarks

The @cc_on statement activates conditional compilation in the scripting engine.

It is strongly recommended that you use the @cc_on statement in a comment, so that browsers that do not support conditional compilation will accept your script as valid syntax:

/*@cc_on*/

...  
(remainder of script)

Alternatively, an @if or @set statement outside of a comment also activates conditional compilation.
Microsoft® JScript®

ceil Method

See Also

Applies To

Description

Returns the smallest integer greater than or equal to its numeric argument.

Syntax

Math.ceil(number)

The number argument is a numeric expression.

Remarks

The return value is an integer value equal to the smallest integer greater than or equal to its numeric argument.
charAt Method

See Also

Description

Returns the character at the specified index.

Syntax

```
strVariable.charAt(index)
"String Literal".charAt(index)
```

The index argument is the zero-based index of the desired character. Valid values are between 0 and the length of the string minus 1.

Remarks

The `charAt` method returns a character value equal to the character at the specified index. The first character in a string is at index 0, the second is at index 1, and so forth. Values of index out of valid range return `undefined`.

The following example illustrates the use of the `charAt` method:

```javascript
function charAtTest(n)
{
    var str = "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
    var s;
```
s = str.charAt(n - 1);
return(s);
}
**charCodeAt Method**

**See Also**  
**Applies To**

---

**Description**

Returns the Unicode encoding of the specified character.

**Syntax**

```
stringObj.charCodeAt(index)
```

The `charCodeAt` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stringObj</code></td>
<td>Required. A <code>String</code> object or literal.</td>
</tr>
<tr>
<td><code>index</code></td>
<td>Required. The zero-based index of the specified character.</td>
</tr>
</tbody>
</table>

**Remarks**

If there is no character at the specified `index`, `NaN` is returned.

The following example illustrates the use of the `charCodeAt` method:

```javascript
function charCodeAtTest(n)
{
    var str = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```
var s;
s = str.charCodeAt(n - 1);
// Return Unicode character code.
return(s);
}
See Also

Description

Causes two expressions to be executed sequentially.

Syntax

`expression1, expression2`

The `,` operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expression1</code></td>
<td>Any <code>expression</code>.</td>
</tr>
<tr>
<td><code>expression2</code></td>
<td>Any expression.</td>
</tr>
</tbody>
</table>

Remarks

The `,` operator causes the expressions on either side of it to be executed in left-to-right order, and obtains the value of the expression on the right. The most common use for the `,` operator is in the increment expression of a `for` loop. For example:

```javascript
for (i = 0; i < 10; i++, j++)
{
    k = i + j;
}
```
The **for** statement only allows a single expression to be executed at the end of every pass through a loop. The `,` operator is used to allow multiple expressions to be treated as a single expression, thereby getting around the restriction.
Description

Causes comments to be ignored by the JScript parser.

Syntax 1

Single-line Comment:

// comment

Syntax 2

Multiline Comment:

/*@cond
comment
@end*/

The comment argument is the text of any comment you want to include in your script.

Syntax 3

//@condStatement

Syntax 4

/*@cond
CondStatement
@end*/

The CondStatement argument is conditional compilation code to be used if conditional compilation is activated. If Syntax 3 is used, there can be no space between the "//" and "@" characters.
Remarks

Use comments to keep parts of a script from being read by the JScript parser. You can use comments to include explanatory remarks in a program.

If Syntax 1 is used, the parser ignores any text between the comment marker and the end of the line. If Syntax 2 is used, it ignores any text between the beginning and end markers.

Syntaxes 3 and 4 are used to support conditional compilation while retaining compatibility with browsers that do not support that feature. These browsers treat those forms of comments as syntaxes 1 and 2 respectively.

The following example illustrates the most common uses of the comment statement:

```javascript
function myfunction(arg1, arg2)
{
    /* This is a multiline comment that can span as many lines as necessary.
    var r;
    // This is a single line comment.
    r = arg1 + arg2; // Sum the two arguments.
    return(r);
}
```
Comparison Operators

See Also

Description

Returns a Boolean value indicating the result of the comparison.

Syntax

\[ expression1 \text{ comparisonoperator } expression2 \]

The Comparison operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( expression1 )</td>
<td>Any \textit{expression}.</td>
</tr>
<tr>
<td>\textit{comparisonoperator}</td>
<td>Any \textit{comparison operator}.</td>
</tr>
<tr>
<td>( expression2 )</td>
<td>Any \textit{expression}.</td>
</tr>
</tbody>
</table>

Remarks

When comparing strings, JScript uses the Unicode character value of the string expression.

The following describes how the different groups of operators behave depending on the types and values of \( expression1 \) and \( expression2 \):

Relational \((<, >, <=, >=)\)

- Attempt to convert both \( expression1 \) and \( expression2 \) into
numbers.

- If both expressions are strings, do a lexicographical string comparison.
- If either expression is NaN, return false.
- Negative zero equals Positive zero.
- Negative Infinity is less than everything including itself.
- Positive Infinity is greater than everything including itself.

**Equality (==, !=)**

- If the types of the two expressions are different, attempt to convert them to string, number, or Boolean.
- NaN is not equal to anything including itself.
- Negative zero equals positive zero.
- `null` equals both `null` and `undefined`.
- Values are considered equal if they are identical strings, numerically equivalent numbers, the same object, identical Boolean values, or (if different types) they can be coerced into one of these situations.
- Every other comparison is considered unequal.

**Identity (===, !==)**

These operators behave identically to the equality operators except no type conversion is done, and the types must be the same to be considered equal.
**compile Method**

**Description**

Compiles a regular expression into an internal format.

**Syntax**

```
rgexp.compile(pattern)
```

The `compile` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rgexp</code></td>
<td>Required. A <strong>Regular Expression</strong> object. Can be a variable name or a literal.</td>
</tr>
<tr>
<td><code>pattern</code></td>
<td>Required. A <strong>string expression</strong> containing a regular expression pattern to be compiled.</td>
</tr>
</tbody>
</table>

**Remarks**

The `compile` method converts `pattern` into an internal format for faster execution. This allows for more efficient use of regular expressions in loops, for example.

The following example illustrates the use of the `compile` method:

```javascript
function CompileDemo()
{
```
var s = "AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPp"
    // Create regular expression
var r = new RegExp("[A-Z]", "g");
var a = s.match(r) // Find matches.
document.write(a);
    // Compile regular expression
r.compile("[a-z]", "g");
var a = s.match(r) // Find matches.
document.write(a);
}
Microsoft® JScript®  Compound Assignment Operators

Addition (+=)
Bitwise AND (&=)
Bitwise OR (|=)
Bitwise XOR (^=)
Division (/=)
Left Shift (<<=)
Modulus (%=)
Multiplication (*=)
Right Shift (>>=)
Subtraction (=-)
Unsigned Right Shift (>>>=)
Description

Returns a new array consisting of a combination of two arrays.

Syntax

array1.concat(array2)

The `concat` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array1</td>
<td>Required. An <code>Array</code> object to concatenate with <code>array2</code>.</td>
</tr>
<tr>
<td>array2</td>
<td>Required. An <code>Array</code> object to concatenate to the end of <code>array1</code>.</td>
</tr>
</tbody>
</table>

Remarks

The `concat` method returns an `Array` object containing the concatenation of `array1` and `array2`.

If an object reference is copied from either `array1` or `array2` to the result, the object reference in the result still points to the same object. Changes to that object are reflected in both arrays.

The following example illustrates the use of the `concat` method:
function ConcatArrayDemo()
{
    var a, b, c;
    a = new Array(0,1,2,3,4);
    b = new Array(5,6,7,8,9);
    c = a.concat(b);
    return(c);
}
**Microsoft® JScript® concat Method (String)**

**See Also**

**Applies To**

**Description**

Returns a **String** object containing the concatenation of two supplied strings.

**Syntax**

`string1.concat(string2)`

The **concat** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>string1</code></td>
<td>Required. The <strong>String</strong> object or literal to concatenate with <code>string2</code>.</td>
</tr>
<tr>
<td><code>string2</code></td>
<td>Required. A <strong>String</strong> object or literal to concatenate to the end of <code>string1</code>.</td>
</tr>
</tbody>
</table>

**Remarks**

The result of the **concat** method is equivalent to: `result = string1 + string2`.

The following example illustrates the use of the **concat** method:

```javascript
function concatDemo()
```
{ 
    var str1 = "ABCDEFGHIJKLM" 
    var str2 = "NOPQRSTUVWXYZ";  
    var s = str1.concat(str2);  
    // Return concatenated string.  
    return(s); 
}
Conditional Compilation

Description

Allows the use of new JScript language features without sacrificing compatibility with browsers that don't support the features.

Remarks

Conditional compilation is activated by using the `@cc_on` statement, or using an `@if` or `@set` statement outside of a comment. Some typical uses for conditional compilation are using new features in JScript, embedding debugging support into a script, and tracing code execution.

It is strongly recommended that conditional compilation code be placed in comments:

```javascript
/*@cc_on */
/*@if (@_jscript_version == 4)
    alert("JScript version 4");
@else @*/
    alert("You need a more recent script engine.");
/*@end */
```
This example uses special comment delimiters that are only used if conditional compilation is activated by the `@cc_on` statement. Scripting engines that do not understand conditional compilation only see the message informing of the need for a new scripting engine.
The following predefined variables are available for conditional compilation. If a variable is not `true`, it is not defined and behaves as `NaN` when accessed.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>@_win32</code></td>
<td><code>true</code> if running on a Win32 system.</td>
</tr>
<tr>
<td><code>@_win16</code></td>
<td><code>true</code> if running on a Win16 system.</td>
</tr>
<tr>
<td><code>@_mac</code></td>
<td><code>true</code> if running on a Apple Macintosh system.</td>
</tr>
<tr>
<td><code>@_alpha</code></td>
<td><code>true</code> if running on a DEC Alpha processor.</td>
</tr>
<tr>
<td><code>@_x86</code></td>
<td><code>true</code> if running on an Intel processor.</td>
</tr>
<tr>
<td><code>@_mc680x0</code></td>
<td><code>true</code> if running on a Motorola 680x0 processor.</td>
</tr>
<tr>
<td><code>@_PowerPC</code></td>
<td><code>true</code> if running on a Motorola PowerPC processor.</td>
</tr>
<tr>
<td><code>@_jscript</code></td>
<td>Always <code>true</code>.</td>
</tr>
<tr>
<td><code>@_jscript_build</code></td>
<td>Contains the build number of the JScript scripting engine.</td>
</tr>
<tr>
<td><code>@_jscript_version</code></td>
<td>Contains the JScript version number in major.minor format.</td>
</tr>
</tbody>
</table>
?: Operator

See Also

Description

Executes one of two expressions depending on a condition.

Syntax

\[ \text{test} \ ? \ \text{expression1} : \text{expression2} \]

The ?: operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>test</td>
<td>Any <a href="#">Boolean expression</a>.</td>
</tr>
<tr>
<td>expression1</td>
<td>An expression executed if test is true.</td>
</tr>
<tr>
<td>expression2</td>
<td>An expression executed if test is false.</td>
</tr>
</tbody>
</table>

Remarks

The ?: operator is a shortcut for an [if...else](#) statement. It is typically used as part of a larger expression where an if...else statement would be awkward. For example:

```javascript
var now = new Date();
var greeting = "Good" + ((now.getHours() > 17) ? "Good evening" : "Good morning");
```
The example creates a string containing "Good evening." if it is after 6pm. The equivalent code using an **if...else** statement would look as follows:

```javascript
var now = new Date();
var greeting = "Good";
if (now.getHours() > 17)
    greeting += " evening."
else
    greeting += " day."
```
**constructor**

**Property**

**Description**

Specifies the function that creates an object.

**Syntax**

```
object.constructor
```

The required `object` argument is the name of an object or function.

**Remarks**

The `constructor` property is a member of the prototype of every object that has a prototype. This includes all *intrinsic JScript objects* except the `Global` and `Math` objects. The `constructor` property contains a reference to the function that constructs instances of that particular object. For example:

```javascript
x = new String("Hi");
if (x.constructor == String)
    // Do something (the condition will be true)
```

or

```javascript
function MyFunc {
    // Body of function.
```
y = new MyFunc;
if (y.constructor == MyFunc)
   // Do something (the condition will be true)
**Description**

Stops the current iteration of a loop, and starts a new iteration.

**Syntax**

```
continue [label];
```

The optional *label* argument specifies the statement to which *continue* applies.

**Remarks**

You can use the *continue* statement only inside a *while*, *do...while*, *for*, or *for...in* loop. Executing the *continue* statement stops the current iteration of the loop and continues program flow with the beginning of the loop. This has the following effects on the different types of loops:

- *while* and *do...while* loops test their condition, and if true, execute the loop again.
- *for* loops execute their increment expression, and if the test expression is true, execute the loop again.
- *for...in* loops proceed to the next field of the specified variable and execute the loop again.

The following example illustrates the use of the *continue* statement:
function skip5()
{
    var s = "", i=0;
    while (i < 10)
    {
        i++;
        // Skip 5
        if (i==5)
        {
            continue;
        }
        s += i;
    }
    return(s);
}
Description

Returns the cosine of a number.

Syntax

Math.cos(number)

The number argument is a numeric expression for which the cosine is sought.

Remarks

The return value is the cosine of its numeric argument.
Description

Enables basic storage and retrieval of dates and times.

Syntax

```javascript
var newDateObj = new Date()
var newDateObj = new Date(dateVal)
var newDateObj = new Date(year, month, date[, hours[, minutes[, seconds[,ms]]]])
```

The `Date` object constructor syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dateVal</code></td>
<td>If a numeric value, <code>dateVal</code> represents the number of milliseconds in <strong>Universal Coordinated Time</strong> between the specified date and midnight January 1, 1970. If a string, <code>dateVal</code> is parsed according to the rules in the <code>parse</code> method. The <code>dateVal</code> argument can also be a VT_DATE value as returned from some ActiveX® objects.</td>
</tr>
<tr>
<td><code>year</code></td>
<td>Required. The full year, for example, 1976 (and not 76).</td>
</tr>
<tr>
<td><strong>month</strong></td>
<td>Required. The month as an integer between 0 and 11 (January to December).</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>date</strong></td>
<td>Required. The date as an integer between 1 and 31.</td>
</tr>
<tr>
<td><strong>hours</strong></td>
<td>Optional. Must be supplied if <em>minutes</em> is supplied. An integer from 0 to 23 (midnight to 11pm) that specifies the hour.</td>
</tr>
<tr>
<td><strong>minutes</strong></td>
<td>Optional. Must be supplied if <em>seconds</em> is supplied. An integer from 0 to 59 that specifies the minutes.</td>
</tr>
<tr>
<td><strong>seconds</strong></td>
<td>Optional. Must be supplied if <em>milliseconds</em> is supplied. An integer from 0 to 59 that specifies the seconds.</td>
</tr>
<tr>
<td><strong>ms</strong></td>
<td>Optional. An integer from 0 to 999 that specifies the milliseconds.</td>
</tr>
</tbody>
</table>

**Remarks**

**A Date** object contains a number representing a particular instant in time to within a millisecond. If the value of an argument is greater than its range or is a negative number, other stored values are modified accordingly. For example, if you specify 150 seconds, JScript redefines that number as two minutes and 30 seconds.

If the number is **NaN**, that indicates that the object does not represent a specific instant of time. If you pass no parameters to the **Date** object, it is initialized to the current time
(UTC). A value must be given to the object before you can use it.

The range of dates that can be represented in a `Date` object is approximately 285,616 years on either side of January 1, 1970.

The `Date` object has two static methods that are called without creating a `Date` object. They are `parse` and `UTC`. 
++ and -- Operators

**See Also**

**Description**

Used to increment or decrement a variable by one.

**Syntax 1**

\[
\begin{align*}
result &= ++\text{variable} \\
result &= --\text{variable} \\
result &= \text{variable}++ \\
result &= \text{variable}--
\end{align*}
\]

**Syntax 2**

\[
\begin{align*}
++\text{variable} \\
--\text{variable} \\
\text{variable}++ \\
\text{variable}--
\end{align*}
\]

The syntax of the ++ and -- operators has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any <strong>variable</strong>.</td>
</tr>
<tr>
<td>variable</td>
<td>Any variable.</td>
</tr>
</tbody>
</table>

**Remarks**

The increment and decrement operators are used as a shortcut to
modify the value stored in a variable. The value of an expression containing one of these operators depends on whether the operator comes before or after the variable:

```javascript
var j, k;
k = 2;
j = ++k;
```

$j$ is assigned the value 3, as the increment occurs before the expression is evaluated.

Contrast the following example:

```javascript
var j, k;
k = 2;
j = k++;
```

Here, $j$ is assigned the value 2, as the increment occurs after the expression is evaluated.


**Microsoft® JScript®**

## description Property

### See Also

**Applies to**

---

**Description**

Returns or sets the descriptive string associated with a specific error.

**Syntax**

```
object.description [= stringexpression]
```

The `description` property syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Any instance of an <code>Error</code> object.</td>
</tr>
<tr>
<td><code>stringexpression</code></td>
<td>A <code>string expression</code> containing a description of the error.</td>
</tr>
</tbody>
</table>

### Remarks

The `description` property contains the error message string associated with a specific error. Use the value contained in this property to alert a user to an error that you can't or don't want to handle.

The following example illustrates the use of the `description` property:

```javascript
try {
```


x = y // Cause an error.
}
catch(var e { // Create local variable e.
    document.write(e) // Prints "[object Error]
    document.write((e.number & 0xFFFF))// Prints 5009
    document.write(e.description) // Prints '"y' is unde
}
Method

Dimensions Method

See Also  Applies To

Description

Returns the number of dimensions in a VBArray.

Syntax

array.dimensions( )

The array argument is a VBArray object.

Remarks

The dimensions method provides a way to retrieve the number of dimensions in a specified VBArray.

The following example consists of three parts. The first part is VBScript code to create a Visual Basic safe array. The second part is JScript code that determines the the number of dimensions in the safe array and the upper bound of each dimension. Both of these parts go into the <HEAD> section of an HTML page. The third part is the JScript code that goes in the <BODY> section to run the other two parts.

<HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
Function CreateVBArray()
    Dim i, j, k
    Dim a(2, 2)
    k = 1
    For i = 0 To 2
        For j = 0 To 2
            a(j, i) = k
            k = k + 1
        Next
    Next
    CreateVBArray = a
End Function

<SCRIPT LANGUAGE="JScript">
!---
function VBArrayTest(vba)
{
    var i, s;
    var a = new VBArray(vba);
    for (i = 1; i <= a.dimensions(); i++)
    {
        s = "The upper bound of dimension ";
        s += i + " is ";
        s += a.ubound(i)+ ".<BR>";
    }
    return(s);
}
-->
</SCRIPT>
</HEAD>

<BODY>
<SCRIPT language="jscript">
document.write(VBAarrayTest(CreateVBAarray()));
</SCRIPT>
</BODY>
do...while Statement

Description

Executes a statement block once, and then repeats execution of the loop until a condition expression evaluates to false.

Syntax

```
do
  statement
while (expression) ;
```

The do...while statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>statement</td>
<td>The statement to be executed if expression is <code>true</code>.</td>
</tr>
<tr>
<td></td>
<td>Can be a compound statement.</td>
</tr>
<tr>
<td>expression</td>
<td>An <code>expression</code> that can be coerced to Boolean <code>true</code> or <code>false</code>. If expression is <code>true</code>, the loop is executed again. If expression is <code>false</code>, the loop is terminated.</td>
</tr>
</tbody>
</table>

Remarks

The value of `expression` is not checked until after the first iteration of the loop, guaranteeing that the loop is executed at least once. Thereafter, it is checked after each succeeding iteration of the loop.
The following code uses the `do...while` statement to iterate the `Drives` collection:

```javascript
function GetDriveList()
{
    var fso, s, n, e, x;
    fso = new ActiveXObject("Scripting.FileSystemObject");
e = new Enumerator(fso.Drives);
s = "";
do
{
    x = e.item();
s = s + x.DriveLetter;
s += " - ";
if (x.DriveType == 3)
    n = x.ShareName;
else if (x.IsReady)
    n = x.VolumeName;
else
    n = "[Drive not ready]";
s += n + "<br>";
e.moveNext();
} while (!e.atEnd());
```
return(s);
}
**E Property**

---

**Description**

Returns Euler's constant, the base of natural logarithms. The E property is approximately equal to 2.718.

**Syntax**

```javascript
var numVar
numVar = Math.E
```
**Microsoft® JScript® Enumerator Object**

**Description**

Enables enumeration of items in a collection.

**Syntax**

```javascript
new Enumerator(collection)
```

The `collection` argument is any collection object.

**Remarks**

Collections differ from arrays in that the members of a collection are not directly accessible. Instead of using indexes, as you would with arrays, you can only move the current item pointer to the first or next element of a collection.

The `Enumerator` object provides a way to access any member of a collection and behaves similarly to the `For...Each` statement in VBScript.

The following code shows the usage of the `Enumerator` object:

```javascript
function ShowDriveList()
{
    var fso, s, n, e, x;
```
fso = new ActiveXObject("Scripting.FileSystemObject");
e = new Enumerator(fso.Drives);
s = "";
for (;!e.atEnd();e.moveNext())
{
    x = e.item();
    s = s + x.DriveLetter;
    s += " - ";
    if (x.DriveType == 3)
    
        n = x.ShareName;
    else if (x.IsReady)
    
        n = x.VolumeName;
    else
    
        n = "[Drive not ready]";
    s += n + "<br>";
}
return(s);
Error Object

See Also

Properties

Description

Contains information about errors.

Syntax

```javascript
var newErrorObj = new Error()
var newErrorObj = new Error(number)
var newErrorObj = new Error(number, description)
```

The Error object constructor syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>number</code></td>
<td>Numeric value assigned to an error. Zero if omitted.</td>
</tr>
<tr>
<td><code>description</code></td>
<td>Brief string that describes an error. Empty string if omitted.</td>
</tr>
</tbody>
</table>

Remarks

Whenever a run-time error occurs, an instance of the Error object is created to describe the error. This instance has two intrinsic properties that contain the description of the error (description property) and the error number (number property).

An error number is a 32-bit value. The upper 16-bit word is the facility code, while the lower word is the actual error code.
Error objects can also be explicitly created, using the syntax shown above, or thrown using the `throw` statement. In both cases, you can add any properties you choose, to expand the capability of the Error object.

Typically, the local variable that's created in a try...catch statement refers to the implicitly created Error object. As a result, you can use the error number and description in any way you choose.

The following example illustrates the use of the implicitly created Error object:

```javascript
try {
    x = y // Cause an error.
} catch(e) {
    // Create local variable e.
    response.write(e) // Prints "[object Error]"
    response.write(e.number & 0xFFFF) // Prints 5009.
    response.write(e.description) // Prints "'y' is undefined"
}
```
Description

Encodes String objects so they can be read on all computers.

Syntax

escape(charstring)

The charstring argument is a String object to be encoded.

Remarks

The escape method returns a new String object (in Unicode format) that contains the contents of charstring. All spaces, punctuation, accented characters, and any other non-ASCII characters are replaced with %xx encoding, where xx is equivalent to the hexadecimal number representing the character. For example, a space is returned as "%20."

Characters with a value greater than 255 are stored using the %uxxxx format.
Description

Evaluates JScript code and executes it.

Syntax

**eval(codestring)**

The *codestring* argument is a **String** object that contains valid JScript code. This string is parsed by the JScript parser and executed.

Remarks

The **eval** function allows dynamic execution of JScript source code. For example, the following code creates a new variable *mydate* that contains a **Date** object:

```javascript
eval("var mydate = new Date();");
```

The code passed to the **eval** method is executed in the same context as the call to the **eval** method.
exec Method

Description

Executes a search for a match in a specified string.

Syntax

\texttt{rgexp.exec(str)}

The \texttt{exec} method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{rgexp}</td>
<td>Required. A \textbf{Regular Expression} object. Can be a variable name or a literal.</td>
</tr>
<tr>
<td>\texttt{str}</td>
<td>Required. The string to perform a search on.</td>
</tr>
</tbody>
</table>

Remarks

The results of an \texttt{exec} method search are placed into an array.

If the \texttt{exec} method does not find a match, it returns \texttt{null}. If it finds one or more matches, the \texttt{exec} method returns an array, and the \texttt{RegExp} object is updated to reflect the results of the search.

The following example illustrates the use of the \texttt{exec} method:

\begin{verbatim}
function ExecDemo()
\end{verbatim}
{
    var s = "AaBbCcDdEeFfGgHhIiJjKkLlMmNn";
    var r = new RegExp("g", "i");
    var a = r.exec(s);
    document.write(a);
    r.compile("g");
    var a = r.exec(s);
    document.write(a);
}
**Description**

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

**Syntax**

```javascript
Math.exp(number)
```

The `number` argument is a numeric expression representing the power of e.

**Remarks**

The return value is $e^{\text{number}}$. The constant $e$ is Euler's constant, approximately equal to 2.178 and `number` is the supplied argument.
fixed Method

See Also

Applies To

Description

Places HTML <TT> tags around text in a String object.

Syntax

strVariable.fixed()
"String Literal".fixed()

Remarks

The following example demonstrates how the fixed method works:

var strVariable = "This is a string object";
strVariable = strVariable.fixed();

The value of strVariable after the last statement is:

<TT>This is a string object</TT>

No checking is done to see if the tag has already been applied to the string.
floor Method

See Also

Applies To

Description

Returns the greatest integer less than or equal to its numeric argument.

Syntax

Math.floor(number)

The number argument is a numeric expression.

Remarks

The return value is an integer value equal to the greatest integer less than or equal to its numeric argument.
See Also

Applies To

Description

Places an HTML &lt;FONT&gt; tag with the COLOR attribute around the text in a String object.

Syntax

```
strVariable.fontcolor(colorval)
"String Literal".fontcolor(colorval)
```

The colorval argument is a string containing a color value. This can either be the hexadecimal value for a color, or the predefined name for a color.

Remarks

The following example demonstrates the fontcolor method:

```
var strVariable = "This is a string";
strVariable = strVariable.fontcolor("red");
```

The value of strVariable after the last statement is:

```
&lt;FONT COLOR="RED">This is a string</FONT&gt
```

Valid predefined color names depend on your JScript host (browser, server, and so forth). They may also vary from version to version of your host. Check your host documentation for more information.
No checking is done to see if the tag has already been applied to the string.
**fontsize Method**

**Description**

Places an HTML `<FONT>` tag with the SIZE attribute around the text in a `String` object.

**Syntax**

```
strVariable.fontsize(intSize)
"String Literal".fontsize(intSize)
```

The `intSize` argument is an integer value that determines the size of the text.

**Remarks**

The following example demonstrates the `fontsize` method:

```
var strVariable = "This is a string";
strVariable = strVariable.fontsize(-1);
```

The value of `strVariable` after the last statement is:

```
<FONT SIZE="-1">This is a string</FONT>
```

Valid integer values depend on your Microsoft JScript host. See your host documentation for more information.

No checking is done to see if the tag has already been applied to the string.
See Also

Description

Executes a block of statements for as long as a specified condition is true.

Syntax

```javascript
for (initialization; test; increment) statement
```

The `for` statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>initialization</code></td>
<td>An expression. This expression is executed only once, before the loop is executed.</td>
</tr>
<tr>
<td><code>test</code></td>
<td>A Boolean expression. If <code>test</code> is <code>true</code>, <code>statement</code> is executed. If <code>test</code> is <code>false</code>, the loop is terminated.</td>
</tr>
<tr>
<td><code>increment</code></td>
<td>An expression. The increment expression is executed at the end of every pass through the loop.</td>
</tr>
<tr>
<td><code>statement</code></td>
<td>The statement to be executed if <code>test</code> is <code>true</code>. Can be a compound <code>statement</code>.</td>
</tr>
</tbody>
</table>
Remarks

You usually use a **for** loop when the loop is to be executed a specific number of times. The following example demonstrates a **for** loop.

```javascript
/* i is set to 0 at start, and is incremented by 1 at the end of each iteration. Loop terminates when i is not less than 10 before a loop iteration. */
var myarray = new Array();
for (i = 0; i < 10; i++)
{
    myarray[i] = i;
}
```
**for...in Statement**

**See Also**

---

**Description**

Executes one or more statements for each *property* of an object, or each element of an array.

**Syntax**

```plaintext
for (variable in [object | array])
    statement
```

The `for` statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>A <em>variable</em> that can be any property of <em>object</em> or any element of <em>array</em>.</td>
</tr>
<tr>
<td>object, array</td>
<td>An object or array over which to iterate.</td>
</tr>
<tr>
<td>statement</td>
<td>The statement or statements to be executed for each property of <em>object</em> or each element of <em>array</em>. Can be a <em>compound statement</em>.</td>
</tr>
</tbody>
</table>

**Remarks**

Before each iteration of a loop, *variable* is assigned the next property of *object* or the next element of *array*. You can then use it in any of the statements inside the loop, exactly as if you were
using the property of object or the element of array.

When iterating over an object, there is no way to determine or control the order in which the members of the object are assigned to variable. Iterating through an array will be performed in element order, that is, 0, 1, 2, ...

The following example illustrates the use of the for ... in statement with an object used as an associative array:

```javascript
function ForInDemo()
{
    // Create some variables.
    var a, key, s = "";
    // Initialize object.
    a = {"a" : "Athens", "b" : "Belgrade", "c" : "Cairo"} // Iterate the properties.
    for (key in a)
    {
        s += a[key] + "<BR;>";
    }
    return(s);
}
```

**Note** Use the enumerator object to iterate members of a collection.
**fromCharCode Method**

**See Also**  
**Applies To**

---

**Description**

Returns a string from a number of Unicode character values.

**Syntax**

```
String.fromCharCode(code1, code2, ..., coden)
```

The `code` argument is the series of Unicode character values to convert into a string.

**Remarks**

A `String` object need not be created before calling `fromCharCode`.

In the following example, `test` contains the string "plain":

```
var test = String.fromCharCode(112, 108, 97, 1
```
Description

Creates a new function.

Syntax 1

```javascript
function functionname( [argname1 [, ... argnameN]] )
{
    body
}
```

Syntax 2

```javascript
var functionname = new Function( [argname1, [... argnameN]],
body );
```

The **Function** object syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>functionname</code></td>
<td>The name of the newly created function</td>
</tr>
<tr>
<td><code>argname1...argnameN</code></td>
<td>An optional list of arguments that the function accepts.</td>
</tr>
<tr>
<td><code>body</code></td>
<td>A string that contains the block of JScript code to be executed when the function is called.</td>
</tr>
</tbody>
</table>
Remarks

The function is a basic data type in JScript. Syntax 1 creates a function value that JScript converts into a Function object when necessary. JScript converts Function objects created by Syntax 2 into function values at the time the function is called.

Syntax 1 is the standard way to create new functions in JScript. Syntax 2 is an alternative form used to create function objects explicitly.

For example, to create a function that adds the two arguments passed to it, you can do it in either of two ways:

Example 1

```javascript
function add(x, y)
{
    return(x + y);
}
```

Example 2

```javascript
var add = new Function("x", "y", "return(x+y)"
```

In either case, you call the function with a line of code similar to the following:

```javascript
add(2, 3);
```

Note  When calling a function, ensure that you always include the parentheses and any required arguments. Calling a function
without parentheses causes the text of the function to be returned instead of the results of the function.
Microsoft® JScript® getItem Method

See Also

Description

Returns the item at the specified location.

Syntax

```
safeArray.getItem(dimension1[, dimension2, ...], dimensionn)
```

The `getItem` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>safeArray</code></td>
<td>Required. A <strong>VBAArray</strong> object.</td>
</tr>
<tr>
<td><code>dimension1, ...</code>, <code>dimensionn</code></td>
<td>Specifies the exact location of the desired element of the VBAArray. <em>n</em> equals the number of dimensions in the VBAArray.</td>
</tr>
</tbody>
</table>

The following example consists of three parts. The first part is VBScript code to create a Visual Basic safe array. The second part is JScript code that iterates the VB safe array and prints out the contents of each element. Both of these parts go into the `<HEAD>` section of an HTML page. The third part is the JScript code that goes in the `<BODY>` section to run the other two parts.

```
<HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
Function CreateVBAArray()
```
Dim i, j, k
Dim a(2, 2)
k = 1
For i = 0 To 2
    For j = 0 To 2
        a(i, j) = k
        document.writeln(k)
        k = k + 1
    Next
    document.writeln("<BR>")
Next
CreateVBArray = a
End Function

--> 
</SCRIPT>
<SCRIPT LANGUAGE="JScript">
<!--
function GetItemTest(vbarray)
{
    var i, j;
    var a = new VBArray(vbarray);
    for (i = 0; i <= 2; i++)
    {
        for (j = 0; j <= 2; j++)
        {
            document.writeln(a.getItem(i, j));
        }
    }

</SCRIPT>
}-->
</SCRIPT>
</HEAD>
<BODY>
<SCRIPT LANGUAGE="JScript">
<!--[CDATA[

    GetItemTest(CreateVBAArray());

-->}
</SCRIPT>
</BODY>
GetObject Function

See Also

Description

Returns a reference to an Automation object from a file.

Syntax

GetObject([pathname] [, class])

The GetObject function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pathname</td>
<td>Optional. Full path and name of the file containing the object to retrieve. If pathname is omitted, class is required.</td>
</tr>
<tr>
<td>class</td>
<td>Optional. Class of the object.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>appname</td>
<td>Required. Name of the application providing the object.</td>
</tr>
<tr>
<td>objectype</td>
<td>Required. Type or class of object to create.</td>
</tr>
</tbody>
</table>

Remarks

Use the GetObject function to access an Automation object from
a file. Assign the object returned by `GetObject` to the object variable. For example:

```javascript
var CADObject;
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`:

```javascript
var LayerObject = GetObject("C:\\CAD\\SCHEF
```

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:
var MyObject;
MyObject = GetObject("C:\DRAWDINGS\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:\DRAWDINGS\SAMPLE.DRW")

---

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

If an object has registered itself as a single-instance object, only
one instance of the object is created, no matter how many times
ActiveXObject 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.
Description

Conditionally executes a group of statements, depending on the value of an expression.

Syntax

```plaintext
@if (condition1)
  text1
[@elif (condition2)
  text2]
[@else
  text3]
@end
```

The `@if` statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition1, condition2</td>
<td>An expression that can be coerced into a <a href="#">Boolean expression</a>.</td>
</tr>
<tr>
<td>text1</td>
<td>Text to be parsed if <code>condition1</code> is <code>true</code>.</td>
</tr>
<tr>
<td>text2</td>
<td>Text to be parsed if <code>condition1</code> is <code>false</code> and <code>condition2</code> is <code>true</code>.</td>
</tr>
<tr>
<td>text3</td>
<td>Text to be parsed if both</td>
</tr>
</tbody>
</table>
text3  condition1 and condition2 are false.

Remarks

When you write an @if statement, you don't have to place each clause on a separate line. You can use multiple @elif clauses, however, all @elif clauses must come before an @else clause.

You commonly use the @if statement to determine which text among several options should be used for text output. For example:

```plaintext
alert(@if (@_win32) "using Windows NT or Windows 95"
```
if...else Statement

See Also

Description

Conditionally executes a group of statements, depending on the value of an expression.

Syntax

```javascript
if (condition)
    statement1
[else
    statement2]
```

The if...else statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition</td>
<td>A <strong>Boolean expression</strong>. If <code>condition</code> is <code>null</code> or <code>undefined</code>, <code>condition</code> is treated as <code>false</code>.</td>
</tr>
<tr>
<td>statement1</td>
<td>The statement to be executed if <code>condition</code> is <code>true</code>. Can be a <strong>compound statement</strong>.</td>
</tr>
<tr>
<td>statement2</td>
<td>The statement to be executed if <code>condition</code> is <code>false</code>. Can be a compound statement.</td>
</tr>
</tbody>
</table>
Remarks

It is generally good practice to enclose statement1 and statement2 in braces ({} for clarity and to avoid inadvertent errors. In the following example, you may intend that the else be used with the first if statement, but it is used with the second one.

```
if (x == 5)
  if (y == 6)
    z = 17;
else
  z = 20;
```

Changing the code in the following manner eliminates any ambiguities:

```
if (x == 5)
  {
    if (y == 6)
      z = 17;
  }
else
  z = 20;
```

Similarly, if you want to add a statement to statement1, and you don't use braces, you can accidentally create an error:

```
if (x == 5)
```
\begin{verbatim}
z = 7;
qu = 42;
else
  z = 19;
\end{verbatim}

In this case, there is a syntax error, because there is more than one statement between the \texttt{if} and \texttt{else} statements. Braces are required around the statements between \texttt{if} and \texttt{else}.
**indexOf Method**

**See Also**

**Applies To**

**Description**

Returns the character position where the first occurrence a substring occurs within a String object.

**Syntax**

```
strVariable.indexOf(substring, startIndex)
"String Literal".indexOf(substring, startIndex)
```

The `indexOf` method syntax has these arguments:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>substring</code></td>
<td>The substring to search for within the String object.</td>
</tr>
<tr>
<td><code>startIndex</code></td>
<td>An optional integer value specifying the index to begin searching within the String object. If omitted, searching begins at the beginning of the string.</td>
</tr>
</tbody>
</table>

**Remarks**

The `indexOf` method returns an integer value indicating the beginning of the substring within the String object. If the substring is not found, a -1 is returned.
If `startIndex` is negative, `startIndex` is treated as zero. If it is larger than the greatest character position index, it is treated as the largest possible index.

Searching is performed from left to right. Otherwise, this method is identical to `lastIndexOf`.

The following example illustrates the use of the `indexOf` method:

```javascript
function IndexDemo(str2)
{
    var str1 = "BABEBIBOBUBABEBE"
    var s = str1.indexOf(str2);
    return(s);
}
```
isNaN Method

Description

Returns a Boolean value that indicates whether a value is the reserved value NaN (not a number).

Syntax

`isNaN(numvalue)`

The `numvalue` argument is the value to be tested against NaN.

Remarks

The `isNaN` function returns `true` if the value is NaN, and `false` otherwise. You typically use this function to test return values from the `parseInt` and `parseFloat` methods.

Alternatively, a variable could be compared to itself. If it compares as unequal, it is NaN. This is because NaN is the only value that is not equal to itself.
**italics Method**

**Microsoft® JScript®**

**Language Reference**

**Version 1**

---

**See Also**

**Applies To**

---

**Description**

Places HTML `<I>` tags around text in a **String** object.

**Syntax**

```
strVariable.italics()
"String Literal".italics()
```

**Remarks**

The following example demonstrates how the **italics** method works:

```
var strVariable = "This is a string";
strVariable = strVariable.italics();
```

The value of `strVariable` after the last statement is:

```
<I>This is a string</I>
```

No checking is done to see if the tag has already been applied to the string.
item Method

See Also

Applies To

Description

Returns the current item in the collection.

Syntax

myEnum.item( )

The myEnum argument is any Enumerator object.

Return Value

The item method returns the current item. If the collection is empty or the current item is undefined, it returns undefined.

Remarks

In following code, the item method is used to return a member of the Drives collection:

```javascript
function ShowDriveList()
{
    var fso, s, n, e, x;
    fso = new ActiveXObject("Scripting.FileSystemObject");
```
e = new Enumerator(fso.Drives);
s = "";
for (; !e.atEnd(); e.moveNext())
{
    x = e.item();
    s = s + x.DriveLetter;
    s += " - ";
    if (x.DriveType == 3)
        n = x.ShareName;
    else if (x.IsReady)
        n = x.VolumeName;
    else
        n = "[Drive not ready]";
    s += n + "<br>");
}
return(s);
**Description**

Returns a **String** object consisting of all the elements of an array concatenated together.

**Syntax**

```
arrayobj.join(separator)
```

The `separator` argument is a **String** object that is used to separate one element of an array from the next in the resulting **String** object. If omitted, the array elements are separated with an empty string.

**Remarks**

The **join** method returns a **String** object that contains each element converted to a string and concatenated together.

The following example illustrates the use of the **join** method:

```javascript
function JoinDemo()
{
    var a, b;
    a = new Array(0,1,2,3,4);
    b = a.join("-");
}
```
return(b);
}
Description

Provides an identifier for a statement.

Syntax

label :
    statement

Labeled statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>A unique identifier used when referring to the labeled statement.</td>
</tr>
<tr>
<td>statement</td>
<td>The statement associated with label. May be a compound statement.</td>
</tr>
</tbody>
</table>

Remarks

Labels are used by the break and continue statements to specify the statement to which the break and continue apply.

In the following statement the continue statement uses a labeled statement to create an array in which the third column of each row contains and undefined value:
function labelDemo()
{
    var a = new Array();
    var i, j, s = "", s1 = ";

    Outer:
    for (i = 0; i < 5; i++)
    {
        Inner:
        for (j = 0; j < 5; j++)
        {
            if (j == 2)
                continue Inner;
            else
                a[i,j] = j + 1;
        }
    }

    for (i = 0; i < 5; i++)
    {
        s = ""
        for (j = 0; j < 5; j++)
        {
            s += a[i,j];
        }
    }
```c
}
s1 += s + "\n";
}
return(s1)
```
Microsoft® JScript®

lastIndexOf

Method

See Also

Applies To

Description

Returns the last occurrence of a substring within a String object.

Syntax

strVariable.lastIndexOf(substring, startIndex)
"String Literal".lastIndexOf(substring, startIndex)

The lastIndexOf method syntax has these arguments:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>substring</td>
<td>The substring to search for within the String object.</td>
</tr>
<tr>
<td>startIndex</td>
<td>An optional integer value specifying the index to begin searching within the String object. If omitted, searching begins at the end of the string.</td>
</tr>
</tbody>
</table>

Remarks

The lastIndexOf method returns an integer value indicating the beginning of the substring within the String object. If the substring is not found, a -1 is returned.
If `startIndex` is negative, `startIndex` is treated as zero. If it is larger than the greatest character position index, it is treated as the largest possible index.

Searching is performed right to left. Otherwise, this method is identical to `indexOf`.

The following example illustrates the use of the `lastIndexOf` method:

```javascript
function lastIndexOfDemo(str2)
{
    var str1 = "BABEBIBOBUBABE"
    var s = str1.lastIndexOf(str2);
    return(s);
}
```
Description

Returns the lowest index value used in the specified dimension of a VBArray.

Syntax

```javascript
safeArray.lbound(dimension)
```

The `lbound` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>safeArray</code></td>
<td>Required. A VBArray object.</td>
</tr>
<tr>
<td><code>dimension</code></td>
<td>Optional. The dimension of the VBArray for which the lower bound index is wanted. If omitted, <code>lbound</code> behaves as if a 1 was passed.</td>
</tr>
</tbody>
</table>

Remarks

If the VBArray is empty, the `lbound` method returns `undefined`. If `dimension` is greater than the number of dimensions in the VBArray, or is negative, the method generates a "Subscript out of range" error.

The following example consists of three parts. The first part is VBScript code to create a Visual Basic safe array. The second part is JScript code that determines the number of dimensions in the safe array and the lower
bound of each dimension. Since the safe array is created in VBScript rather than Visual Basic, the lower bound will always be zero. Both of these parts go into the <HEAD> section of an HTML page. The third part is the JScript code that goes in the <BODY> section to run the other two parts.

```html
<HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
Function CreateVBArray()
    Dim i, j, k
    Dim a(2, 2)
    k = 1
    For i = 0 To 2
        For j = 0 To 2
            a(j, i) = k
            k = k + 1
        Next
    Next
    CreateVBArray = a
End Function
-->  
</SCRIPT>

<SCRIPT LANGUAGE="JScript">
<!--
function VBArrayTest(vba)
{
```
```javascript
var i, s;
var a = new VBAArray(vba);
for (i = 1; i <= a.dimensions(); i++)
{
    s = "The lower bound of dimension ";
    s += i + " is ";
    s += a.lbound(i) + "<BR>";
    return(s);
}

</SCRIPT>
</HEAD>

<BODY>
<SCRIPT language="jscript">
document.write(VBAArrayTest(CreateVBAArray()));
</SCRIPT>
</BODY>
```
**length Property** (Array)

**Description**

Returns an integer value one higher than the highest element defined in an array.

**Syntax**

```javascript
numVar = arrayObj.length
```

**Remarks**

As the elements in an array do not have to be contiguous, the `length` property is not necessarily the number of elements in the array. For example, in the following array definition,

```javascript
var my_array = new Array();
my_array[0] = "Test";
my_array[6] = "Another Test";
```

`my_array.length` contains 7, not 2:

If a value smaller than its previous value is assigned to the `length` property, the array is truncated, and any elements with array indexes equal to or greater than the new value of the `length` property are lost.

If a value larger than its previous value is assigned to the `length` property, the array is expanded, and any new elements created have the value `undefined`. 
The following example illustrates the use of the `length` property:

```javascript
function LengthDemo()
{
    var a, l;
    a = new Array(0,1,2,3,4);
    l = a.length;
    return(l);
}
```
Description

Returns the number of arguments defined for a function.

Syntax

```
functionname.length
```

The `functionname` argument is required and is the name of the function in question.

Remarks

The `length` property of a function is initialized by the scripting engine to the number of arguments in the function's definition when an instance of the function is created.

What happens when a function is called with a number of arguments different from the value of its `length` property depends on the function.

The following example illustrates the use of the `length` property:

```
function ArgTest(a, b)
{
    var i, s = "The ArgTest function expected ";
```
var numargs = ArgTest.arguments.length;
var expargs = ArgTest.length;
if (expargs < 2)
  s += expargs + " argument. ";
else
  s += expargs + " arguments. ";
if (numargs < 2)
  s += numargs + " was passed.";
else
  s += numargs + " were passed.";
return (s);
**Microsoft® JScript® length Property**

*(String)*

**See Also**

**Applies To**

---

**Description**

Returns the length of a *String* object.

**Syntax**

```
strVariable.length
"String Literal".length
```

**Remarks**

The *length* property contains an integer that indicates the number of characters in the *String* object. The last character in the *String* object has an index of *length* - 1.
Description

Places an HTML anchor with an HREF attribute around the text in a String object.

Syntax

```javascript
strVariable.link(linkstring)
"String Literal".link(linkstring)
```

The `linkstring` argument is the text that you want to place in the HREF attribute of the HTML anchor.

Remarks

Call the `link` method to create a hyperlink out of a String object. The following is an example of how the method accomplishes this:

```javascript
var strVariable = "This is a hyperlink";
strVariable = strVariable.link("http://www.microsoft.com"
```

The value of `strVariable` after the last statement is:

```
<A HREF="http://www.microsoft.com">This is
```

No checking is done to see if the tag has already been applied to the string.
**LN2 Property**

**Description**

Returns the natural logarithm of 2.

**Syntax**

```javascript
var numVar
numVar = Math.LN2
```

**Syntax**

The **LN2** property is approximately equal to 0.693.
 LN10 Property  

Description

Returns the natural logarithm of 10.

Syntax

```javascript
var numVar
numVar = Math.LN10
```

Remarks

The LN10 property is approximately equal to 2.302.
**log Method**

**Description**

Returns the natural logarithm of a number.

**Syntax**

```
Math.log(number)
```

The `number` argument is a numeric expression for which the natural logarithm is sought.

**Return Value**

The return value is the natural logarithm of `number`. The base is $e$. 

**LOG2E Property**

**Description**

Returns the base-2 logarithm of e, Euler's constant.

**Syntax**

```javascript
var varName
varName = objName.LOG2E
```

**Remarks**

The **LOG2E** property, a constant, is approximately equal to 1.442.
**LOG10E**

**Property**

**Description**

Returns the base-10 logarithm of $e$, Euler's constant.

**Syntax**

```javascript
var varName
varName = objName.LOG10E
```

**Remarks**

The **LOG10E** property, a constant, is approximately equal to 0.434.
See Also

Description

Performs a logical conjunction on two expressions.

Syntax

\[ result = expression1 \&\& expression2 \]

The \&\& operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any variable.</td>
</tr>
<tr>
<td>expression1</td>
<td>Any expression.</td>
</tr>
<tr>
<td>expression2</td>
<td>Any expression.</td>
</tr>
</tbody>
</table>

Remarks

If, and only if, both expressions evaluate to \textbf{True}, \textit{result} is \textbf{True}. If either expression evaluates to \textbf{False}, \textit{result} is \textbf{False}.

For information on when a run-time error is generated by the \&\& operator, see the Operator Behavior table.

JScript uses the following rules for converting non-Boolean values to Boolean values:

- All objects are considered true.
• Strings are considered false if, and only if, they are empty.
• **null** and **undefined** are considered false.
• Numbers are false if, and only if, they are zero.
### ! Operator

**See Also**

**Description**

Performs logical negation on an expression.

**Syntax**

\[ \text{result} = !\text{expression} \]

The ! operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any <strong>variable</strong>.</td>
</tr>
<tr>
<td>expression</td>
<td>Any <strong>expression</strong>.</td>
</tr>
</tbody>
</table>

**Remarks**

The following table illustrates how `result` is determined.

<table>
<thead>
<tr>
<th>If <code>expression is</code></th>
<th>Then <code>result is</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

All unary operators, such as the ! operator, evaluate expressions as follows:

- If applied to **undefined** or **null** expressions, a **run-time error** is raised.
- Objects are converted to strings.
- Strings are converted to numbers if possible. If not, a run-time error is raised.
- Boolean values are treated as numbers (0 if false, 1 if true).

The operator is applied to the resulting number.

For the ! operator, if expression is nonzero, result is zero. If expression is zero, result is 1.
Description

Performs a logical disjunction on two expressions.

Syntax

\[ result = expression1 \ || \ expression2 \]

The || operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any variable.</td>
</tr>
<tr>
<td>expression1</td>
<td>Any expression.</td>
</tr>
<tr>
<td>expression2</td>
<td>Any expression.</td>
</tr>
</tbody>
</table>

Remarks

If either or both expressions evaluate to True, result is True. The following table illustrates how result is determined:

<table>
<thead>
<tr>
<th>If expression1 is</th>
<th>And expression2 is</th>
<th>The result is</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>
For information on when a run-time error is generated by the `&&` operator, see the Operator Behavior table.

JScript uses the following rules for converting non-Boolean values to Boolean values:

- All objects are considered true.
- Strings are considered false if and only if they are empty.
- `null` and `undefined` are considered false.
- Numbers are false if, and only if, they are 0.
**Math Object**

**Description**

An intrinsic object that provides basic mathematics functionality and constants.

**Syntax**

```javascript
Math.[{property | method}]
```

**Remarks**

The **Math** object cannot be created using the **new** operator, and gives an error if you attempt to do so. It is created by the scripting engine when the engine is loaded. All of its methods and properties are available to your script at all times.
**Description**

Returns the greater of two supplied numeric expressions.

**Syntax**

\[ retval = \textbf{Math.max}(number1, number2) \]

The \textbf{max} method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{retval}</td>
<td>The greater of \textit{number1} or \textit{number2}.</td>
</tr>
<tr>
<td>\textit{number1}</td>
<td>A \textit{numeric expression} to be compared to \textit{number2}.</td>
</tr>
<tr>
<td>\textit{number2}</td>
<td>A numeric value to be compared to \textit{number1}.</td>
</tr>
</tbody>
</table>
**min Method**

**See Also**

**Applies To**

**Description**

Returns the lesser of two supplied numbers.

**Syntax**

\[ \text{retVal} = \text{Math.min}(\text{number1}, \text{number2}) \]

The `min` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>retVal</code></td>
<td>The lesser of <code>number1</code> or <code>number2</code>.</td>
</tr>
<tr>
<td><code>number1</code></td>
<td>A numeric expression to be compared to <code>number2</code>.</td>
</tr>
<tr>
<td><code>number2</code></td>
<td>A numeric value to be compared to <code>number1</code>.</td>
</tr>
</tbody>
</table>
**Description**

Resets the current item in the collection to the first item.

**Syntax**

```
myEnum.moveFirst()
```

The `myEnum` argument is any `Enumerator` object.

**Remarks**

If there are no items in the collection, the current item is set to `undefined`.

In following example, the `moveFirst` method is used to begin evaluating members of the `Drives` collection from the beginning of the list:

```javascript
function ShowFirstAvailableDrive()
{
    var fso, s, e, x;
    fso = new ActiveXObject("Scripting.FileSystemObject");
e = new Enumerator(fso.Drives);
e.moveFirst();
```
s = "";
do {
    x = e.item();
    if (x.IsReady)
    {
        s = x.DriveLetter + ":";
        break;
    }
    else
    {
        if (e.atEnd())
        {
            s = "No drives are available";
            break;
        }
        e.moveNext();
    }
} while (!e.atEnd());
return(s);
**moveNext Method**

**See Also**

**Applies To**

---

**Description**

Moves the current item to the next item in the collection.

**Syntax**

```javascript
myEnum.moveNext()
```

The `myEnum` argument is any `Enumerator` object.

**Remarks**

If the enumerator is at the end of the collection or the collection is empty, the current item is set to `undefined`.

In following example, the `moveNext` method is used to move to the next drive in the `Drives` collection:

```javascript
function ShowDriveList()
{
    var fso, s, n, e, x;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    e = new Enumerator(fso.Drives);
    s = ""
;
for (; !e.atEnd(); e.moveNext())
{
    x = e.item();
    s = s + x.DriveLetter;
    s += " - ";
    if (x.DriveType == 3)
        n = x.ShareName;
    else if (x.IsReady)
        n = x.VolumeName;
    else
        n = "[Drive not ready]";
    s += n + "<br>";
}
return(s);
**NaN Property**

**See Also**

**Applies To**

---

**Description**

A special value that indicates an arithmetic expression returned a value that was not a number.

**Syntax**

```
Number.NaN
```

The `number` argument is the `Number` object.

**Remarks**

The `Number` object does not have to be created before the `NaN` property can be accessed.

`NaN` does not compare equal to any value, including itself. To test if a value is equivalent to `NaN`, use the `isNaN` function.
new Operator

See Also

Description

Creates a new object.

Syntax

`new constructor[(arguments)]`

The constructor argument calls object's constructor. The parentheses can be omitted if the constructor takes no arguments.

Remarks

The `new` operator performs the following tasks:

1. It creates an object with no members.
2. It calls the constructor for that object, passing a pointer to the newly created object as the `this` pointer.

The constructor then initializes the object according to the arguments passed to the constructor.

These are examples of valid uses of the `new` operator:

```javascript
my_object = new Object;
my_array = new Array();
```
my_date = new Date("Jan 5 1996")
**number Property**

**Description**

Returns or sets the numeric value associated with a specific error. The Error object’s default property is `number`.

**Syntax**

`object.number [ = errornumber ]`

The `number` property syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Any instance of the Error object.</td>
</tr>
<tr>
<td><code>errornumber</code></td>
<td>An integer representing an error.</td>
</tr>
</tbody>
</table>

**Remarks**

An error number is a 32-bit value. The upper 16-bit word is the facility code, while the lower word is the actual error code.

The following example illustrates the use of the `number` property:

```javascript
try {
    x = y                                  // Cause an error.
}  
```
catch(var e) { // Create local var:
    document.write(e) // Prints "[object Error]"
    document.write(e.number>>16 & 0x1FFF) // F
    document.write(e.number & 0xFFFF) // Prints "5009"
    document.write(e.description) // Prints "'y"
**Object Object**

**Description**

Provides functionality common to all JScript objects.

**Syntax**

```javascript
new Object([value])
```

The optional `value` argument is used to convert a `primitive` data type (number, Boolean, string, or function) into an object. If omitted, an object with no contents is created.

**Remarks**

The **Object** object is contained in all other JScript objects--all of its methods and properties are available in all other objects. The methods can be redefined in user-defined objects, and are called by JScript at appropriate times. The **toString** method is an example of a frequently redefined **Object** method.

In this language reference, the description of each **Object** method includes both default and object-specific implementation information for the **intrinsic JScript objects**.
Operators in JScript are evaluated in a particular order. This order is known as the operator precedence. The following table lists the operators in highest to lowest precedence order. Operators with the same precedence are evaluated in left to right order in the expression.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>. [] ()</td>
<td>Field access, array indexing, and function calls</td>
</tr>
<tr>
<td>++ -- - ~ ! delete new typeof void</td>
<td>Unary operators, return data type, object creation, undefined values</td>
</tr>
<tr>
<td>* / %</td>
<td>Multiplication, division, modulo division</td>
</tr>
<tr>
<td>+ - +</td>
<td>Addition, subtraction, string concatenation</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt; &gt;&gt;</td>
<td>Bit shifting</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;= instanceof</td>
<td>Less than, less than or equal, greater than, greater than or equal, instanceof</td>
</tr>
<tr>
<td>== != === !==</td>
<td>Equality, inequality, identity, nonidentity</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>Conditional</td>
</tr>
<tr>
<td>= OP=</td>
<td>Assignment, assignment with operation</td>
</tr>
<tr>
<td>,</td>
<td>Multiple evaluation</td>
</tr>
</tbody>
</table>

Parentheses are used to alter the order of evaluation. The expression within
parentheses is fully evaluated before its value is used in the remainder of the statement.

An operator with higher precedence is evaluated before one with lower precedence. For example:

\[ z = 78 \times (96 + 3 + 45) \]

There are five operators in this expression: =, *, (), +, and +. According to precedence, they are evaluated in the following order: (), *, +, +, =.

1. Evaluation of the expression within the parentheses is first: There are two addition operators, and they have the same precedence: 96 and 3 are added together and 45 is added to that total, resulting in a value of 144.

2. Multiplication is next: 78 and 144 are multiplied, resulting in a value of 10998.

3. Assignment is last: 11232 is assigned into z.
Description

Returns the ratio of the circumference of a circle to its diameter, approximately 3.141592653589793.

Syntax

```javascript
var numVar
numVar = Math.PI
```

The `PI` property, a constant, is approximately equal to 3.14159.
Description

Returns the value of a base expression taken to a specified power.

Syntax

Math.pow(base, exponent)

The `pow` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>base</code></td>
<td>The base value of the expression.</td>
</tr>
<tr>
<td><code>exponent</code></td>
<td>The exponent value of the expression.</td>
</tr>
</tbody>
</table>

Remarks

In the following example, a `numeric expression` equal to `base^{exponent}` returns 1000.

Math.pow(10,3);
**Description**

Returns a reference to the prototype for a class of objects.

**Syntax**

```
objectname.prototype
```

The `objectname` argument is the name of an object.

**Remarks**

Use the `prototype` property to provide a base set of functionality to a class of objects. New instances of an object "inherit" the behavior of the prototype assigned to that object.

For example, say you want to add a method to the `Array` object that returns the value of the largest element of the array. To do this, declare the function, add it to `Array.prototype`, and then use it.

```javascript
function array_max() {
    var i, max = this[0];
    for (i = 1; i < this.length; i++)
```
{  
    if (max < this[i])
        max = this[i];
}

return max;
}

Array.prototype.max = array_max;
var x = new Array(1, 2, 3, 4, 5, 6);
var y = x.max( );

After this code is executed, y contains the largest value in the array x, or 6.

All intrinsic JScript objects have a prototype property that is read-only. Functionality may be added to the prototype, as in the example, but the object may not be assigned a different prototype. However, user-defined objects may be assigned a new prototype.

The method and property lists for each intrinsic object in this language reference indicate which ones are part of the object's prototype, and which are not.
random Method

See Also

Applies To

Description

Returns a pseudorandom number between 0 and 1.

Syntax

Math.random( )

Remarks

The pseudorandom number generated is between 0 and 1 inclusive. The random number generator is seeded automatically when JScript is first loaded.
RegExp Object

Description

Stores information on regular expression pattern searches.

Syntax

`RegExp.propertyname`

The `propertyname` argument is one of the `RegExp` object properties.

Remarks

The `RegExp` object cannot be created directly, but is always available for use. Its properties have `undefined` as their value until a successful regular expression search has been completed.

The following example illustrates the use of the `RegExp` object:

```javascript
function matchDemo()
{
    var s;
    var re = new RegExp("d(b+)d","ig");
    var str = "cdbBdbsbdbdz";
    var arr = re.exec(str);
```
s = "$1 contains: " + RegExp.$1 + "<BR>";
s += "$2 contains: " + RegExp.$2 + "<BR>";
s += "$3 contains: " + RegExp.$3;
return(s);
Regular Expression Object

Description

Contains a regular expression pattern.

Syntax 1

```
var regularexpression = /pattern/[switch]
```

Syntax 2

```
var regularexpression = new RegExp("pattern",["switch"])
```

The regular expression object syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern</td>
<td>Required. The regular expression pattern to use. If you use Syntax 1, delimit the pattern by &quot;/&quot; characters. If you use Syntax 2, enclose the pattern in quotation marks.</td>
</tr>
<tr>
<td>Switch</td>
<td>Optional. Enclose switch in quotation marks if you use Syntax 2. Available switches are:</td>
</tr>
<tr>
<td></td>
<td>• i (ignore case)</td>
</tr>
<tr>
<td></td>
<td>• g (global search for all occurrences of pattern)</td>
</tr>
<tr>
<td></td>
<td>• gi (global search, ignore case)</td>
</tr>
</tbody>
</table>
Remarks

**Regular Expression** objects store patterns used when searching strings for character combinations. After the **Regular Expression** object is created, it is either passed to a string method, or a string is passed to one of the regular expression methods. Information about the most recent search performed is stored in the **RegExp** object.

Use Syntax 1 when you know the search string ahead of time. Use Syntax 2 when the search string is changing frequently, or is unknown, such as strings taken from user input.

The *pattern* argument is compiled into an internal format before use. For Syntax 1, *pattern* is compiled as the script is loaded. For Syntax 2, *pattern* is compiled just before use, or when the **compile** method is called.
Description

Special characters and sequences are used in writing patterns for regular expressions. The following table describes these characters and includes short examples showing how the characters are used.

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\</td>
<td>Marks the next character as either a special character or a literal. For example, &quot;n&quot; matches the character &quot;n&quot;. &quot;\n&quot; matches a newline character. The sequence &quot;&quot; matches &quot;&quot; and &quot;(&quot; matches &quot;)&quot;.</td>
</tr>
<tr>
<td>^</td>
<td>Matches the beginning of input.</td>
</tr>
<tr>
<td>$</td>
<td>Matches the end of input.</td>
</tr>
<tr>
<td>*</td>
<td>Matches the preceding character zero or more times. For example, &quot;zo*&quot; matches either &quot;z&quot; or &quot;zoo&quot;.</td>
</tr>
<tr>
<td>+</td>
<td>Matches the preceding character one or more times. For example, &quot;zo+&quot; matches &quot;zoo&quot; but not &quot;z&quot;.</td>
</tr>
<tr>
<td>?</td>
<td>Matches the preceding character zero or one time. For example, &quot;a?ve?&quot; matches the &quot;ve&quot; in &quot;never&quot;.</td>
</tr>
<tr>
<td>.</td>
<td>Matches any single character except a newline character.</td>
</tr>
<tr>
<td>(pattern)</td>
<td>Matches pattern and remembers the match. The matched substring can be retrieved from the resulting Matches collection, using Item [0]...[n]. To match parentheses characters ( ), use &quot;(&quot; or &quot;)&quot;.</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>{n}</td>
<td>n is a nonnegative integer. Matches exactly n times. For example, &quot;o{2}&quot; does not match the &quot;o&quot; in &quot;Bob,&quot; but matches the first two o's in &quot;foooood&quot;.</td>
</tr>
<tr>
<td>{n,}</td>
<td>n is a nonnegative integer. Matches at least n times. For example, &quot;o{2,}&quot; does not match the &quot;o&quot; in &quot;Bob&quot; and matches all the o's in &quot;foooood&quot;. &quot;o{1,}&quot; is equivalent to &quot;o+&quot;. &quot;o{0,}&quot; is equivalent to &quot;o*&quot;.</td>
</tr>
<tr>
<td>{n,m}</td>
<td>m and n are nonnegative integers. Matches at least n and at most m times. For example, &quot;o{1,3}&quot; matches the first three o's in &quot;foooood&quot;. &quot;o{0,1}&quot; is equivalent to &quot;o?&quot;.</td>
</tr>
<tr>
<td>[xyz]</td>
<td>A character set. Matches any one of the enclosed characters. For example, &quot;[abc]&quot; matches the &quot;a&quot; in &quot;plain&quot;.</td>
</tr>
<tr>
<td>[^xyz]</td>
<td>A negative character set. Matches any character not enclosed. For example, &quot;[^abc]&quot; matches the &quot;p&quot; in &quot;plain&quot;.</td>
</tr>
<tr>
<td>[a-z]</td>
<td>A range of characters. Matches any character in the specified range. For example, &quot;[a-z]&quot; matches any lowercase alphabetic character in the range &quot;a&quot; through &quot;z&quot;.</td>
</tr>
<tr>
<td>[^m-z]</td>
<td>A negative range characters. Matches any character not in the specified range. For example, &quot;[^m-z]&quot; matches any character not in the range &quot;m&quot; through &quot;z&quot;.</td>
</tr>
<tr>
<td>\b</td>
<td>Matches a word boundary, that is, the position between a word and a space. For example, &quot;er\b&quot; matches the &quot;er&quot; in &quot;never&quot; but not the &quot;er&quot; in &quot;verb&quot;.</td>
</tr>
<tr>
<td>\B</td>
<td>Matches a nonword boundary. &quot;ea*r\B&quot; matches the &quot;ear&quot; in &quot;never early&quot;.</td>
</tr>
<tr>
<td>\d</td>
<td>Matches a digit character. Equivalent to [0-9].</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>\D</td>
<td>Matches a nondigit character. Equivalent to [^0-9].</td>
</tr>
<tr>
<td>\f</td>
<td>Matches a form-feed character.</td>
</tr>
<tr>
<td>\n</td>
<td>Matches a newline character.</td>
</tr>
<tr>
<td>\r</td>
<td>Matches a carriage return character.</td>
</tr>
<tr>
<td>\s</td>
<td>Matches any white space including space, tab, form-feed, etc. Equivalent to &quot;[ \f\n\r\t\v]&quot;.</td>
</tr>
<tr>
<td>\S</td>
<td>Matches any nonwhite space character. Equivalent to &quot;[^A-Za-z0-9_]&quot;.</td>
</tr>
<tr>
<td>\t</td>
<td>Matches a tab character.</td>
</tr>
<tr>
<td>\v</td>
<td>Matches a vertical tab character.</td>
</tr>
<tr>
<td>\w</td>
<td>Matches any word character including underscore. Equivalent to &quot;[A-Za-z0-9_]&quot;.</td>
</tr>
<tr>
<td>\W</td>
<td>Matches any nonword character. Equivalent to &quot;[^A-Za-z0-9_]&quot;.</td>
</tr>
<tr>
<td>\num</td>
<td>Matches num, where num is a positive integer. A reference back to remembered matches. For example, &quot;(.\1&quot; matches two consecutive identical characters.</td>
</tr>
<tr>
<td>\n</td>
<td>Matches n, where n is an octal escape value. Octal escape values must be 1, 2, or 3 digits long. For example, &quot;\11&quot; and &quot;\011&quot; both match a tab character. &quot;\0011&quot; is the equivalent of &quot;\011&quot; &amp; &quot;1&quot;. 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.</td>
</tr>
<tr>
<td>\xn</td>
<td>Matches n, where n is a hexadecimal escape value. Hexadecimal escape values must be exactly two digits long. For example, &quot;\x41&quot; matches &quot;A&quot;. &quot;\x041&quot; is equivalent to &quot;\x04&quot; &amp; &quot;1&quot;. Allows ASCII codes to be used in regular expressions.</td>
</tr>
</tbody>
</table>
**Description**

Exits from the current function and returns a value from that function.

**Syntax**

```
return [expression];
```

The `expression` argument is the value to be returned from the function. If omitted, the function does not return a value.

**Remarks**

You use the `return` statement to stop execution of a function and return the value of `expression`. If `expression` is omitted, or no `return` statement is executed from within the function, the `expression` that called the current function is assigned the value `undefined`.

The following example illustrates the use of the `return` statement:

```javascript
function myfunction(arg1, arg2)
{
    var r;
    r = arg1 * arg2;
}
```
return(r);
}
Microsoft® JScript® reverse Method

Language Reference
Version 2

See Also
Applies To

Description

Returns an Array object with the elements reversed.

Syntax

arrayobj.reverse()

Remarks

The reverse method reverses the elements of an Array object in place. It does not create a new Array object during execution.

If the array is not contiguous, the reverse method creates elements in the array that fill the gaps in the array. Each of these created elements has the value undefined.

The following example illustrates the use of the reverse method:

```javascript
function ReverseDemo()
{
    var a, l;
    a = new Array(0,1,2,3,4);
    l = a.reverse();
    return(l);
}
```
}

__________________________
**Description**

Returns a supplied numeric expression rounded to the nearest integer.

**Syntax**

```
Math.round(number)
```

The `number` argument is the value to be rounded to the nearest integer.

**Remarks**

If the decimal portion of `number` is 0.5 or greater, the return value is equal to the smallest integer greater than `number`. Otherwise, `round` returns the largest integer less than or equal to `number`. 
ScriptEngine Function

See Also

Description

Returns a string representing the scripting language in use.

Syntax

ScriptEngine( )

Return Values

The ScriptEngine function can return any of the following strings:

<table>
<thead>
<tr>
<th>String</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JScript</td>
<td>Indicates that Microsoft JScript is the current scripting engine.</td>
</tr>
<tr>
<td>VBA</td>
<td>Indicates that Microsoft Visual Basic® for Applications is the current scripting engine.</td>
</tr>
<tr>
<td>VBScript</td>
<td>Indicates that Microsoft Visual Basic Scripting Edition is the current scripting engine.</td>
</tr>
</tbody>
</table>

Remarks
The following code illustrates the use of the `ScriptEngine` function:

```javascript
function GetScriptEngineInfo()
{
    var s;
    s = ""; // Build string with necessary info.
    s += ScriptEngine() + " Version ";
    s += ScriptEngineMajorVersion() + ".";
    s += ScriptEngineMinorVersion() + ".";
    s += ScriptEngineBuildVersion();
    return(s);
}
```
ScriptEngineBuildVersion Function

See Also

Description

Returns the build version number of the scripting engine in use.

Syntax

ScriptEngineBuildVersion( )

Return Values

The return value corresponds directly to the version information contained in the dynamic-link library (DLL) for the scripting language in use.

Remarks

The following code illustrates the use of the ScriptEngineBuildVersion function:

```javascript
function GetScriptEngineInfo()
{
    var s;
    s = ""; // Build string with necessary info.
    s += ScriptEngine() + " Version ";
    s += ScriptEngineMajorVersion() + ".";
    s += ScriptEngineMinorVersion() + ".";
    s += ScriptEngineBuildVersion();
}
return(s);
}

ScriptEngineMajorVersion Function

Description

Returns the major version number of the scripting engine in use.

Syntax

ScriptEngineMajorVersion( )

Return Values

The return value corresponds directly to the version information contained in the dynamic-link library(DLL) for the scripting language in use.

Remarks

The following code illustrates the use of the ScriptEngineMajorVersion function:

```javascript
function GetScriptEngineInfo()
{
    var s;
    s = ""; // Build string with necessary info.
    s += ScriptEngine() + " Version ";
```
s += ScriptEngineMajorVersion() + ".";
s += ScriptEngineMinorVersion() + ".";
s += ScriptEngineBuildVersion();
return(s);
}
Microsoft® JScript®

ScriptEngineMinorVersion Function

See Also

Description

Returns the minor version number of the scripting engine in use.

Syntax

`ScriptEngineMinorVersion()`

Return Values

The return value corresponds directly to the version information contained in the dynamic-link library (DLL) for the scripting language in use.

Remarks

The following code illustrates the use of the `ScriptEngineMinorVersion` function:

```javascript
function GetScriptEngineInfo()
{
    var s;
    s = ""; // Build string with necessary info.
    s += ScriptEngine() + " Version ";
```
s += ScriptEngineMajorVersion() + ".";

s += ScriptEngineMinorVersion() + ".";

s += ScriptEngineBuildVersion();

return(s);
}
# @set Statement

## Description

Creates variables used with conditional compilation statements.

## Syntax

```plaintext
@set @varname = term
```

The `@set` statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>varname</code></td>
<td>Valid JScript variable name. Must be preceded by an &quot;@&quot; character at all times.</td>
</tr>
<tr>
<td><code>term</code></td>
<td>Zero or more unary operators followed by a constant, conditional compilation variable, or parenthesized expression.</td>
</tr>
</tbody>
</table>

## Remarks

Numeric and Boolean variables are supported for conditional compilation. Strings are not. Variables created using `@set` are generally used in conditional compilation statements, but can be used anywhere in JScript code.

Examples of variable declarations look like this:

```plaintext
@set @myvar1 = 12
```
@set @myvar2 = (@myvar1 * 20)
@set @myvar3 = @_jscript_version

The following operators are supported in parenthesized expressions:

- ! ~
- * / %
- + -
- << >> >>>
- < <= > >=
- == != === !=
- & ^ |
- && ||

If a variable is used before it has been defined, its value is NaN. NaN can be checked for using the @if statement:

@if (@newVar != @newVar)
...

This works because NaN is the only value not equal to itself.
**Description**

Returns the sine of a number.

**Syntax**

```javascript
Math.sin(number)
```

The `number` argument is a numeric expression for which the sine is sought.

**Remarks**

The return value is the sine of its numeric argument.
Microsoft® JScript® slice Method

(Array)

Description

Returns a section of an array.

Syntax

```
arrayObj.slice(start, [end])
```

The slice method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrayObj</td>
<td>Required. An Array object.</td>
</tr>
<tr>
<td>start</td>
<td>Required. The zero-based index of the beginning of the specified portion of arrayObj.</td>
</tr>
<tr>
<td>end</td>
<td>Optional. The zero-based index of the end of the specified portion of arrayObj.</td>
</tr>
</tbody>
</table>

Remarks

The slice method returns an Array object containing the specified portion of arrayObj.

The slice method copies up to, but not including, the element indicated by end. If negative, end indicates an offset from the end of arrayObj. In addition, it is not zero-based. If omitted, extraction continues to the end of arrayObj.
In the following example, all but the last element of \textit{myArray} is copied into \textit{newArray}:

\[
\text{newArray} = \text{myArray}.\text{slice}(0, -1)
\]

If an object reference is copied from \textit{arrayObj} to the result, the object reference in the result still points to the same object. Changes to that object are reflected in both arrays.
slice Method
(String)

See Also
Applies To

Description

Returns a section of a string.

Syntax

stringObj.slice(start, [end])

The slice method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stringObj</td>
<td>Required. A String object or literal.</td>
</tr>
<tr>
<td>start</td>
<td>Required. The zero-based index of the beginning of the specified portion of stringObj.</td>
</tr>
<tr>
<td>end</td>
<td>Optional. The zero-based index of the end of the specified portion of stringObj.</td>
</tr>
</tbody>
</table>

Remarks

The slice method returns a String object containing the specified portion of stringObj.

If negative, end indicates an offset from the end of stringObj. In addition, it is not zero-based. If omitted, extraction continues to the end of stringObj.

In the example that follows, the two uses of the slice method return the
same thing. Negative one in the second example points to the last character in str1 as the ending point:

```go
str1.slice(0)
str2.slice(0,-1)
```
**Description**

Places HTML `<SMALL>` tags around text in a `String` object.

**Syntax**

```javascript
strVariable.small()
"String Literal".small()
```

**Remarks**

The example that follows demonstrates how the `small` method works:

```javascript
var strVariable = "This is a string";
strVariable = strVariable.small();
```

The value of `strVariable` after the last statement is:

`<SMALL>This is a string</SMALL>`

No checking is done to see if the tag has already been applied to the string.
**Microsoft® JScript® sort Method**

**See Also**

**Applies To**

---

**Description**

Returns an **Array** objec with the elements sorted.

**Syntax**

```javascript
arrayobj.sort(sortfunction)
```

The `sortfunction` argument is the name of the function used to determine the order of the elements. If omitted, the elements are sorted in ascending, ASCII character order.

**Remarks**

The `sort` method sorts the **Array** object in place; no new **Array** object is created during execution.

If you supply a function in the `sortfunction` argument, it must return one of the following values:

- A negative value if the first argument passed is less than the second argument.
- Zero if the two arguments are equivalent.
- A positive value if the first argument is greater than the second argument.

The following example illustrates the use of the `sort` method:

```javascript
function SortDemo()
```
{  
    var a, l;
    a = new Array("X", "y", "d", "Z", "v", "m", "r");
    l = a.sort();
    return(l);
}

source Property

Description

Returns a copy of the text of the regular expression pattern. Read-only.

Syntax

`rgexp.source`

The `rgexp` argument is a Regular expression object. It can be a variable name or a literal.

The following example illustrates the use of the `source` property:

```javascript
function SourceDemo(re, s) {
    var s1;
    // Test string for existence of regular expressic
    if (re.test(s))
        s1 = " contains ";
    else
        s1 = " does not contain ";
    // Get the text of the regular expression itself.
}````
return(s + s1 + re.source);
}
**Description**

Returns the square root of a number.

**Syntax**

```
Math.sqrt(number)
```

The `number` argument is a numeric expression.

**Remarks**

If `number` is negative, the return value is zero.
**SQRT1_2 Property**

**See Also**

**Applies To**

**Description**

Returns the square root of 0.5, or one divided by the square root of 2.

**Syntax**

```javascript
var numVar
numVar = Math.SQRT1_2
```

**Remarks**

The **SQRT1_2** property, a constant, is approximately equal to 0.707.
### SQRT2 Property

#### See Also

#### Applies To

#### Description

Returns the square root of 2.

#### Syntax

```javascript
var numVar
numVar = Math.SQRT2
```

#### Syntax

The `SQRT2` property, a constant, is approximately equal to 1.414.
**strike Method**

**Description**

Places HTML `<STRIKE>` tags around text in a `String` object.

**Syntax**

```
strVariable.strike()
"String Literal".strike()
```

**Remarks**

The following example demonstrates how the `strike` method works:

```javascript
var strVariable = "This is a string object";
strVariable = strVariable.strike();
```

The value of `strVariable` after the last statement is:

```html
<STRIKE>This is a string object</STRIKE>
```

No checking is done to see if the tag has already been applied to the string.
**String Object**

**Description**

Allows manipulation and formatting of text strings and determination and location of substrings within strings.

**Syntax**

```
StringObj.[method]
"String Literal".[method]
```

**Remarks**

String objects can be created implicitly using string literals. String objects created in this fashion (referred to as standard strings) are treated differently than String objects created using the new operator. All string literals share a common, global string object. So, if a property is added to a string literal, it is available to all standard string objects:

```javascript
var alpha, beta;
alpha = "This is a string";
beta = "This is also a string";
alpha.test = 10;
```

In this example, test is now defined for beta and all future string literals. In the following
example, however, added properties are treated differently:

```javascript
var gamma, delta;
gamma = new String("This is a string");
delta = new String("This is also a string");
gamma.test = 10;
```

In this case, `test` is not defined for `delta`. Each `String` object declared as a `new String` object has its own set of members. This is the only case where `String` objects and string literals are handled differently.
See Also

Applies To

Description

Places HTML <SUB> tags around text in a String object.

Syntax

    strVariable.sub( )
    "String Literal".sub( )

Remarks

The following example demonstrates how the sub method works:

    var strVariable = "This is a string object"
    strVariable = strVariable.sub( );

The value of strVariable after the last statement is:

    <SUB>This is a string object</SUB>

No checking is done to see if the tag has already been applied to the string.
**substr Method**

See Also | Applies To
---|---

**Description**

Returns a substring beginning at a specified location and having a specified length.

**Syntax**

`stringvar.substr(start [, length ])`

The `substr` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stringvar</code></td>
<td>Required. A string literal or <code>String</code> object from which the substring is extracted.</td>
</tr>
<tr>
<td><code>start</code></td>
<td>Required. The starting position of the desired substring. The index of the first character in the string is zero.</td>
</tr>
<tr>
<td><code>length</code></td>
<td>Optional. The number of characters to include in the returned substring.</td>
</tr>
</tbody>
</table>

**Remarks**

If `length` is zero or negative, an empty string is returned. If not specified, the substring continues to the end of `stringvar`.

The following example illustrates the use of the `substr` method:
function SubstrDemo()
{
  var s, ss;
  var s = "The quick brown fox jumped over the 
  ss = s.substr(16, 3);
  // Returns "fox".
  return(ss);
}

See Also

Applies To

Description

Returns the substring at the specified location within a `String` object.

Syntax

```
strVariable.substring(start, end)
"String Literal".substring(start, end)
```

The `substring` method syntax has these arguments:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>start</code></td>
<td>The zero-based index indicating the beginning of the substring.</td>
</tr>
<tr>
<td><code>end</code></td>
<td>The zero-based index indicating the end of the substring.</td>
</tr>
</tbody>
</table>

Remarks

The `substring` method returns a `String` object containing the substring derived from the original object.

The `substring` method uses the lower of `start` and `end` as the beginning point of the substring. For example, `strvar.substring(0, 3)` and `strvar.substring(3, 0)` return the same substring.

The only exception to this is for negative parameters. If the first parameter is less than zero, it is treated as zero. If the second parameter is negative, it is set to the value of the
The length of the substring is equal to the absolute value of the difference between start and end. For example, the length of the substring returned in `strvar.substring(0, 3)` and `strvar.substring(3, 0)` is three.

Finally, start and end can be strings. If so, these strings are coerced into integers if possible. If not, the value of the parameter is treated as zero.

The following example illustrates the use of the `substring` method:

```javascript
function SubstringDemo()
{
    var s, ss;
    var s = "The quick brown fox jum
    ss = s.substring(16, 19);
    return(ss);
}
```
Description

Used to find the difference between two numbers or to indicate the negative value of a numeric expression.

Syntax 1

\[ \text{result} = \text{number1} - \text{number2} \]

Syntax 2

\[-\text{number} \]

The - operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any numeric variable.</td>
</tr>
<tr>
<td>number</td>
<td>Any numeric expression.</td>
</tr>
<tr>
<td>number1</td>
<td>Any numeric expression.</td>
</tr>
<tr>
<td>number2</td>
<td>Any numeric expression.</td>
</tr>
</tbody>
</table>

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.

For information on when a run-time error is generated by Syntax 1, see the Operator Behavior table.

For Syntax 2, as for all unary operators, expressions are evaluated as follows:

- If applied to undefined or null expressions, a run-time error is raised.
- Objects are converted to strings.
- Strings are converted to numbers if possible. If not, a run-time error is raised.
- Boolean values are treated as numbers (0 if false, 1 if true).

The operator is applied to the resulting number. In Syntax 2, if the resulting number is nonzero, result is equal to the resulting number with its sign reversed. If the resulting number is zero, result is zero.
sup Method

Description

Places HTML <SUP> tags around text in a String object.

Syntax

```
strVariable.sup()
"String Literal".sup()
```

Remarks

The following example demonstrates how the sup method works:

```
var strVariable = "This is a string object";
strVariable = strVariable.sup();
```

The value of strVariable after the last statement is:

```
<SUP>This is a string object</SUP>
```

No checking is done to see if the tag has already been applied to the string.
See Also

Description

Enables the execution of one or more statements when a specified expression's value matches a label.

Syntax

```javascript
switch (expression) {
    case label :
        statementlist
    case label :
        statementlist
...
    default :
        statementlist
}
```

The `switch` statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expression</code></td>
<td>The expression to be evaluated.</td>
</tr>
<tr>
<td><code>label</code></td>
<td>An identifier to be matched against <code>expression</code>. If <code>label === expression</code>, execution starts with the <code>statementlist</code> immediately after the colon, and continues until it encounters either a <code>break</code> statement, which is optional, or the end of the <code>switch</code> statement.</td>
</tr>
</tbody>
</table>
One or more statements to be executed.

Remarks

Use the **default** clause to provide a statement to be executed if none of the label values matches *expression*. It can appear anywhere within the **switch** code block.

Zero or more *label* blocks may be specified. If no *label* matches the value of *expression*, and a **default** case is not supplied, no statements are executed.

Execution flows through a switch statement as follows:

1. Evaluate *expression* and look at *label* in order until a match is found.

2. If a *label* value equals *expression*, execute its accompanying **statementlist**.
   Continue execution until a **break** statement is encountered, or the **switch** statement ends. This means that multiple *label* blocks are executed if a **break** statement is not used.

3. If no *label* equals *expression*, go to the **default** case. If there is no **default** case, go to last step.

4. Continue execution at the statement following the end of the **switch** code block.

The following example tests an object for its type:

```javascript
function MyObject() {
    ...
}
switch (object.constructor) {
```
case Date:
...
case Number:
...
case String:
...
case MyObject:
...
default:
...
}
Description

Returns the tangent of a number.

Syntax

Math.tan(number)

The number argument is a numeric expression for which the tangent is sought.

Remarks

The return value is the tangent of number.
Description

Returns a Boolean value that indicates whether or not a pattern exists in a searched string.

Syntax

```
rgexp.test(str)
```

The `test` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rgexp</code></td>
<td>Required. A Regular Expression object. Can be a variable name or a literal.</td>
</tr>
<tr>
<td><code>str</code></td>
<td>Required. The string to test a search on.</td>
</tr>
</tbody>
</table>

Remarks

The `test` method checks to see if a pattern exists within a string and returns `true` if so, and `false` otherwise.

The `RegExp` object is not modified by the `test` method.

The following example illustrates the use of the `test` method:

```
function TestDemo(re, s)
```
var s1;
// Test string for existence of regular expression
if (re.test(s))
  s1 = " contains ";
else
  s1 = " does not contain ";
// Get text of the regular expression itself.
return(s + s1 + re.source);


See Also

Description

Refers to the current object.

Syntax

this.property

Remarks

The this keyword is typically used in object constructors to refer to the current object. In the following example, this refers to the newly created Car object, and assigns values to three properties:

```javascript
function Car(color, make, model)
{
    this.color = color;
    this.make = make;
    this.model = model;
}
```

For client versions of JScript, this refers to the window object if used outside of the context of any other object.
See Also

Applies To

Description

Returns a standard JScript array converted from a VBArray.

Syntax

```javascript
safeArray.toArray()
```

The `safeArray` argument is a `VBArray` object.

Remarks

The conversion translates the multidimensional VBArray into a single dimensional JScript array. Each successive dimension is appended to the end of the previous one. For example, a VBArray with three dimensions and three elements in each dimension is converted into a JScript array as follows:

Suppose the VBArray contains: (1, 2, 3), (4, 5, 6), (7, 8, 9). After translation, the JScript array contains: 1, 2, 3, 4, 5, 6, 7, 8, 9.

There is currently no way to convert a JScript array into a VBArray.

The following example consists of three parts. The first part is VBScript code to create a Visual Basic safe array. The second part is JScript code that converts the VB safe array to a JScript array. Both of these parts go into the `<HEAD>` section of an HTML page. The third part is the JScript code that goes in the `<BODY>` section to run the other two parts.
Function CreateVBArray()
    Dim i, j, k
    Dim a(2, 2)
    k = 1
    For i = 0 To 2
        For j = 0 To 2
            a(j, i) = k
            document.writeln(k)
            k = k + 1
        Next
        document.writeln("<BR>")
    Next
    CreateVBArray = a
End Function
-->
{ 
    document.writeln(b[i]);
}

--> 
</SCRIPT>
</HEAD>
<BODY;>
<SCRIPT LANGUAGE="JScript"> 
<!--
    VBArrayTest(CreateVBArray());
--> 
</SCRIPT>
</BODY>
**Microsoft® JScript® toLowerCase Method**

**See Also**

**Applies To**

---

**Description**

Returns a string where all alphabetic characters have been converted to lowercase.

**Syntax**

```
strVariable.toLowerCase()
"String Literal".toLowerCase()
```

**Remarks**

The `toLowerCase` method has no effect on nonalphabetic characters.

The following example demonstrates the effects of the `toLowerCase` method:

```
var strVariable = "This is a STRING"
strVariable = strVariable.toLowerCase()
```

The value of `strVariable` after the last statement is:

```
this is a string object
```
**Description**

Returns a string representation of an object.

**Syntax**

`objectname.toString([radix])`

The `toString` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>objectname</code></td>
<td>Required. An object for which a string representation is sought.</td>
</tr>
<tr>
<td><code>radix</code></td>
<td>Optional. Specifies a radix for converting numeric values to strings.</td>
</tr>
</tbody>
</table>

**Remarks**

The `toString` method is a member of all built-in JScript objects. How it behaves depends on the object type:

<table>
<thead>
<tr>
<th>Object</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>Elements of an <code>Array</code> are converted to strings. The resulting strings are concatenated, separated by commas.</td>
</tr>
<tr>
<td>Boolean</td>
<td>If the Boolean value is <code>true</code>, returns &quot;true&quot;. Otherwise, returns &quot;false&quot;</td>
</tr>
</tbody>
</table>
| Function | Returns a string returned of the following form, where functionname is the name of the function whose toString method was called:
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>function functionname( ) { [native code] }</td>
</tr>
<tr>
<td>Number</td>
<td>Returns the textual representation of the number.</td>
</tr>
<tr>
<td>String</td>
<td>Returns the value of the String object.</td>
</tr>
<tr>
<td>Default</td>
<td>Returns &quot;[object objectname]&quot;, where objectname is the name of the object type.</td>
</tr>
</tbody>
</table>

The following example illustrates the use of the toString method with a radix argument:

function CreateRadixTable ()
{
    var s1, s2, s3, x;
    document.write("Hex Dec Bin<BR>");
    for (x = 0; x < 16; x++)
    {
        switch(x)
        {
            case 0 :
                s1 = "    ";
                s2 = "   ";
                s3 = "  ";
                break;
            case 1 :
s1 = "   ";
s2 = "   ";
s3 = "   ";
break;
case 2:
  s3 = "   ";
  break;
case 3:
  s3 = "   ";
  break;
case 4:
  s3 = "   ";
  break;
case 5:
  s3 = "   ";
  break;
case 6:
  s3 = "   ";
  break;
case 7:
  s3 = "   ";
  break;
case 8:
s3 = "" ;
break;
case 9 :
s3 = "";
break;
default:
s1 = "    ";
s2 = "";
s3 = "    ";
}
document.write(" ", x.toString(16), s1, x.toString(16), s2, " ", x.toString(16), s3, " ");
**toUpperCase Method**

**Description**

Returns a string where all alphabetic characters have been converted to uppercase.

**Syntax**

```
strVariable.toUpperCase()
"String Literal".toUpperCase()
```

**Remarks**

The `toUpperCase` method has no effect on nonalphabetic characters.

The following example demonstrates the effects of the `toUpperCase` method:

```
var strVariable = "This is a STRING"
strVariable = strVariable.toUpperCase()
```

The value of `strVariable` after the last statement is:

**THIS IS A STRING OBJECT**
See Also

Description

Returns a string that identifies the data type of an expression.

Syntax

```
typeof [ ( ] expression [ ) ] ;
```

The `expression` argument is any `expression` for which type information is sought.

Remarks

The `typeof` operator returns type information as a string. There are six possible values that `typeof` returns: "number," "string," "boolean," "object," "function," and "undefined."

The parentheses are optional in the `typeof` syntax.
Description

Returns the highest index value used in the specified dimension of the VBArray.

Syntax

safeArray.ubound(dimension)

The **ubound** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>safeArray</td>
<td>Required. A <strong>VBArray</strong> object.</td>
</tr>
<tr>
<td>dimension</td>
<td>Optional. The dimension of the VBArray for which the higher bound index is wanted. If omitted, <strong>ubound</strong> behaves as if a 1 was passed.</td>
</tr>
</tbody>
</table>

Remarks

If the VBArray is empty, the **ubound** method returns **undefined**. If **dim** is greater than the number of dimensions in the VBArray, or is negative, the method generates a "Subscript out of range" error.

The following example consists of three parts. The first part is VBScript code to create a Visual Basic safe array. The second part is JScript code that determines the the number of dimensions in the safe array and the
upper bound of each dimension. Both of these parts go into the <HEAD> section of an HTML page. The third part is the JScript code that goes in the <BODY> section to run the other two parts.

<HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
Function CreateVBArray()
    Dim i, j, k
    Dim a(2, 2)
    k = 1
    For i = 0 To 2
        For j = 0 To 2
            a(j, i) = k
            k = k + 1
        Next
    Next
    CreateVBArray = a
End Function
-->
</SCRIPT>

<SCRIPT LANGUAGE="JScript">
<!--
function VBArrayTest(vba)
{
    var i, s;
    var a = new VBArray(vba);

}
for (i = 1; i <= a.dimensions(); i++)
{
    s = "The upper bound of dimension ";
    s += i + " is ";
    s += a.ubound(i) + "<BR>";
    return(s);
}

-->
</SCRIPT>
</HEAD>

<BODY>
<SCRIPT language="jscript">
    document.write(VBArrayTest(CreateVBArrray()));
</SCRIPT>
</BODY>
Microsoft® JScript® **unescape Method**

**See Also**

**Applies To**

---

**Description**

Decodes **String** objects encoded with the **escape** method.

**Syntax**

```
unescape(charstring)
```

The `charstring` argument is a **String** object to be decoded.

**Remarks**

The **unescape** method returns a new **String** object that contains the contents of `charstring`. All characters encoded with the %xx hexadecimal form are replaced by their **ASCII** character set equivalents.

Characters encoded in %uxxxx format (Unicode characters) are replaced with the Unicode character with hexadecimal encoding xxxx.
>>> Operator

See Also

Description

Performs an unsigned right shift of the bits in an expression.

Syntax

\[ \text{result} = \text{expression1} \gggg \text{expression2} \]

The \( \gggg \) operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any variable.</td>
</tr>
<tr>
<td>expression1</td>
<td>Any expression.</td>
</tr>
<tr>
<td>expression2</td>
<td>Any expression.</td>
</tr>
</tbody>
</table>

Remarks

The \( \gggg \) operator shifts the bits of \( \text{expression1} \) right by the number of bits specified in \( \text{expression2} \). Zeros are filled in from the left. Digits shifted off the right are discarded. For example:

```javascript
var temp
temp = -14 >>> 2
```

The variable \( temp \) has a value of 1073741820 as \( -14 \) (11111111 11111111 11110010 in binary) shifted right two bits.
equals 1073741820 (00111111 11111111 11111111 11111100 in binary).

For information on when a run-time error is generated by the >>> operator, see the Operator Behavior table.
valueOf Method

See Also

Applies To

Description

Returns the primitive value of the specified object.

Syntax

\[ object.valueOf( ) \]

The \textit{object} argument is any JScript object.

Remarks

The \texttt{valueOf} method is defined differently for each intrinsic JScript object.

<table>
<thead>
<tr>
<th>Object</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>The elements of the array are converted into strings, and the strings are concatenated together, separated by commas. This behaves the same as the \texttt{Array.toString} and \texttt{Array.join} methods.</td>
</tr>
<tr>
<td>Boolean</td>
<td>The Boolean value.</td>
</tr>
<tr>
<td>Date</td>
<td>The stored time value in milliseconds since midnight, January 1, 1970 UTC.</td>
</tr>
<tr>
<td>Function</td>
<td>The function itself.</td>
</tr>
<tr>
<td>Number</td>
<td>The numeric value.</td>
</tr>
<tr>
<td>Object</td>
<td>The object itself. This is the default.</td>
</tr>
</tbody>
</table>
The `Math` object does not have a `valueOf` method.
**var Statement**

---

### See Also

### Description

Declares a **variable**.

### Syntax

```
var variable [ = value ] [, variable2 [ = value2 ], ...]
```

The `var` statement syntax has the following parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>variable</code>, <code>variable2</code></td>
<td>The names of the variables being declared.</td>
</tr>
<tr>
<td><code>value</code>, <code>value2</code></td>
<td>The initial value assigned to the variable.</td>
</tr>
</tbody>
</table>

### Remarks

Use the `var` statement to declare variables. These variables can be assigned values at declaration or later in your script. Examples of declaration follow:

```
var index;
var name = "Thomas Jefferson";
var answer = 42, counter, numpages = 10;
```
VBArray Object

See Also | Methods | Properties

Description

Provides access to Visual Basic safe arrays.

Syntax

```javascript
new VBArray(safeArray)
```

The `safeArray` is a `VBArray` value.

Remarks

VBArrays are read-only, and cannot be created directly. The `safeArray` argument must have obtained a `VBArray` value before being passed to the `VBArray` constructor. This can only be done by retrieving the value from an existing ActiveX or other object.

VBArrays can have multiple dimensions. The indices of each dimension can be different. The `dimensions` method retrieves the number of dimensions in the array; the `lbound` and `ubound` methods retrieve the range of indices used by each dimension.

The following example consists of three parts. The first part is VBScript code to create a Visual Basic safe array. The second part is JScript code that converts the VB safe array to a JScript array. Both of these parts go into the `<HEAD>` section of an HTML page. The third part is the JScript code that goes in the `<BODY>` section to run the other two parts.
<HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
Function CreateVBArray()
    Dim i, j, k
    Dim a(2, 2)
    k = 1
    For i = 0 To 2
        For j = 0 To 2
            a(j, i) = k
            document.writeln(k)
            k = k + 1
        Next
        document.writeln("<BR>")
    Next
    CreateVBArray = a
End Function
-->  
</SCRIPT>
<SCRIPT LANGUAGE="JScript">
<!--
function VBArrayTest(vbarray)
{
    var a = new VBArray(vbarray);
    var b = a.toArray();
    var i;
    for (i = 0; i < 9; i++)
```
{ }
  document.writeln(b[i]);
}

-->
</SCRIPT>
</HEAD>
<BODY;>
<SCRIPT LANGUAGE="JScript">
<!--
    VBArrayTest(CreateVBArray());
-->
</SCRIPT>
</BODY>
See Also

Description

Prevents an expression from returning a value.

Syntax

```
void expression
```

The `expression` argument is any valid JScript `expression`.

Remarks

The `void` operator evaluates its expression, and returns `undefined`. It is most useful in situations where you want an expression evaluated but do not want the results visible to the remainder of the script.
**while Statement**

**Description**

Executes a statement until a specified condition is **false**.

**Syntax**

```
while (expression)  
statement
```

The **while** statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A <strong>Boolean expression</strong> checked before each iteration of the loop. If expression is <strong>true</strong>, the loop is executed. If expression is <strong>false</strong>, the loop is terminated.</td>
</tr>
<tr>
<td>statement</td>
<td>The statement to be executed if expression is <strong>true</strong>. Can be a <strong>compound statement</strong>.</td>
</tr>
</tbody>
</table>

**Remarks**

The **while** statement checks *expression* before a loop is first executed. If *expression* is **false** at this time, the loop is never
executed.

The following example illustrates the use of the `while` statement:

```javascript
function BreakTest(breakpoint)
{
    var i = 0;
    while (i < 100)
    {
        if (i == breakpoint)
            break;
        i++;  
    }
    return(i);
}
```
with Statement

See Also

Description

Establishes the default object for a statement.

Syntax

```
with (object)
statement
```

The `with` statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>The new default object.</td>
</tr>
<tr>
<td><code>statement</code></td>
<td>The statement for which <code>object</code> is the default object. Can be a <a href="#">compound statement</a>.</td>
</tr>
</tbody>
</table>

Remarks

The `with` statement is commonly used to shorten the amount of code that you have to write in certain situations. In the example that follows, notice the repeated use of `Math`:

```
x = Math.cos(3 * Math.PI) + Math.sin(Math.LN10)
y = Math.tan(14 * Math.E)
```
When you use the `with` statement, your code becomes shorter and easier to read:

```python
with (Math) {
    x = cos(3 * PI) + sin (LN10)
    y = tan(14 * E)
}
```
## JScript Syntax Errors

<table>
<thead>
<tr>
<th>Error Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Invalid procedure call or argument</td>
</tr>
<tr>
<td>6</td>
<td>Overflow</td>
</tr>
<tr>
<td>7</td>
<td>Out of memory</td>
</tr>
<tr>
<td>9</td>
<td>Subscript out of range</td>
</tr>
<tr>
<td>10</td>
<td>This array is fixed or temporarily locked</td>
</tr>
<tr>
<td>11</td>
<td>Division by zero</td>
</tr>
<tr>
<td>13</td>
<td>Type mismatch</td>
</tr>
<tr>
<td>14</td>
<td>Out of string space</td>
</tr>
<tr>
<td>17</td>
<td>Can't perform requested operation</td>
</tr>
<tr>
<td>28</td>
<td>Out of stack space</td>
</tr>
<tr>
<td>35</td>
<td>Sub or Function not defined</td>
</tr>
<tr>
<td>48</td>
<td>Error in loading DLL</td>
</tr>
<tr>
<td>51</td>
<td>Internal error</td>
</tr>
<tr>
<td>52</td>
<td>Bad file name or number</td>
</tr>
<tr>
<td>53</td>
<td>File not found</td>
</tr>
<tr>
<td>54</td>
<td>Bad file mode</td>
</tr>
<tr>
<td>55</td>
<td>File already open</td>
</tr>
<tr>
<td>57</td>
<td>Device I/O error</td>
</tr>
<tr>
<td>58</td>
<td>File already exists</td>
</tr>
<tr>
<td>61</td>
<td>Disk full</td>
</tr>
<tr>
<td>62</td>
<td>Input past end of file</td>
</tr>
<tr>
<td>67</td>
<td>Too many files</td>
</tr>
<tr>
<td>68</td>
<td>Device unavailable</td>
</tr>
<tr>
<td>70</td>
<td>Permission denied</td>
</tr>
<tr>
<td>71</td>
<td>Disk not ready</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>74</td>
<td>Can't rename with different drive</td>
</tr>
<tr>
<td>75</td>
<td>Path/File access error</td>
</tr>
<tr>
<td>76</td>
<td>Path not found</td>
</tr>
<tr>
<td>91</td>
<td>Object variable or With block variable not set</td>
</tr>
<tr>
<td>92</td>
<td>For loop not initialized</td>
</tr>
<tr>
<td>94</td>
<td>Invalid use of Null</td>
</tr>
<tr>
<td>322</td>
<td>Can't create necessary temporary file</td>
</tr>
<tr>
<td>424</td>
<td>Object required</td>
</tr>
<tr>
<td>429</td>
<td>Automation server can't create object</td>
</tr>
<tr>
<td>430</td>
<td>Class doesn't support Automation</td>
</tr>
<tr>
<td>432</td>
<td>File name or class name not found during Automation operation</td>
</tr>
<tr>
<td>438</td>
<td>Object doesn't support this property or method</td>
</tr>
<tr>
<td>440</td>
<td>Automation error</td>
</tr>
<tr>
<td>445</td>
<td>Object doesn't support this action</td>
</tr>
<tr>
<td>446</td>
<td>Object doesn't support named arguments</td>
</tr>
<tr>
<td>447</td>
<td>Object doesn't support current locale setting</td>
</tr>
<tr>
<td>448</td>
<td>Named argument not found</td>
</tr>
<tr>
<td>449</td>
<td>Argument not optional</td>
</tr>
<tr>
<td>450</td>
<td>Wrong number of arguments or invalid property assignment</td>
</tr>
<tr>
<td>451</td>
<td>Object not a collection</td>
</tr>
<tr>
<td>453</td>
<td>Specified DLL function not found</td>
</tr>
<tr>
<td>458</td>
<td>Variable uses an Automation type not supported in JScript</td>
</tr>
<tr>
<td>462</td>
<td>The remote server machine does not exist or is unavailable</td>
</tr>
<tr>
<td>501</td>
<td>Cannot assign to variable</td>
</tr>
<tr>
<td>502</td>
<td>Object not safe for scripting</td>
</tr>
<tr>
<td>503</td>
<td>Object not safe for initializing</td>
</tr>
<tr>
<td>504</td>
<td>Object not safe for creating</td>
</tr>
<tr>
<td>507</td>
<td>An exception occurred</td>
</tr>
<tr>
<td>5000</td>
<td>Cannot assign to 'this'</td>
</tr>
<tr>
<td>5001</td>
<td>Number expected</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>5002</td>
<td>Function expected</td>
</tr>
<tr>
<td>5003</td>
<td>Cannot assign to a function result</td>
</tr>
<tr>
<td>5004</td>
<td>Cannot index object</td>
</tr>
<tr>
<td>5005</td>
<td>String expected</td>
</tr>
<tr>
<td>5006</td>
<td>Date object expected</td>
</tr>
<tr>
<td>5007</td>
<td>Object expected</td>
</tr>
<tr>
<td>5008</td>
<td>Illegal assignment</td>
</tr>
<tr>
<td>5009</td>
<td>Undefined identifier</td>
</tr>
<tr>
<td>5010</td>
<td>Boolean expected</td>
</tr>
<tr>
<td>5011</td>
<td>Can't execute code from a freed script</td>
</tr>
<tr>
<td>5012</td>
<td>Object member expected</td>
</tr>
<tr>
<td>5013</td>
<td>VBArray expected</td>
</tr>
<tr>
<td>5014</td>
<td>JScript object expected</td>
</tr>
<tr>
<td>5015</td>
<td>Enumerator object expected</td>
</tr>
<tr>
<td>5016</td>
<td>Regular Expression object expected</td>
</tr>
<tr>
<td>5017</td>
<td>Syntax error in regular expression</td>
</tr>
<tr>
<td>5018</td>
<td>Unexpected quantifier</td>
</tr>
<tr>
<td>5019</td>
<td>Expected ')' in regular expression</td>
</tr>
<tr>
<td>5020</td>
<td>Expected ')' in regular expression</td>
</tr>
<tr>
<td>5021</td>
<td>Invalid range in character set</td>
</tr>
<tr>
<td>5022</td>
<td>Exception thrown and not caught</td>
</tr>
<tr>
<td>5023</td>
<td>Function does not have a valid prototype object</td>
</tr>
</tbody>
</table>
## JScript Run-time Errors

<table>
<thead>
<tr>
<th>Error Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Out of memory</td>
</tr>
<tr>
<td>1002</td>
<td>Syntax error</td>
</tr>
<tr>
<td>1003</td>
<td>Expected ':;'</td>
</tr>
<tr>
<td>1004</td>
<td>Expected ';;'</td>
</tr>
<tr>
<td>1005</td>
<td>Expected '('</td>
</tr>
<tr>
<td>1006</td>
<td>Expected ')'</td>
</tr>
<tr>
<td>1007</td>
<td>Expected ']'</td>
</tr>
<tr>
<td>1008</td>
<td>Expected '{'</td>
</tr>
<tr>
<td>1009</td>
<td>Expected '}'</td>
</tr>
<tr>
<td>1010</td>
<td>Expected identifier</td>
</tr>
<tr>
<td>1011</td>
<td>Expected '='</td>
</tr>
<tr>
<td>1012</td>
<td>Expected '/'</td>
</tr>
<tr>
<td>1013</td>
<td>Invalid number</td>
</tr>
<tr>
<td>1014</td>
<td>Invalid character</td>
</tr>
<tr>
<td>1015</td>
<td>Unterminated string constant</td>
</tr>
<tr>
<td>1016</td>
<td>Unterminated comment</td>
</tr>
<tr>
<td>1018</td>
<td>'return' statement outside of function</td>
</tr>
<tr>
<td>1019</td>
<td>Can't have 'break' outside of loop</td>
</tr>
<tr>
<td>1020</td>
<td>Can't have 'continue' outside of loop</td>
</tr>
<tr>
<td>1023</td>
<td>Expected hexadecimal digit</td>
</tr>
<tr>
<td>1024</td>
<td>Expected 'while'</td>
</tr>
<tr>
<td>1025</td>
<td>Label redefined</td>
</tr>
<tr>
<td>1026</td>
<td>Label not found</td>
</tr>
<tr>
<td>1027</td>
<td>'default' can only appear in a 'switch' statement</td>
</tr>
<tr>
<td>1028</td>
<td>Expected identifier or string</td>
</tr>
<tr>
<td>Line</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>1029</td>
<td>Expected '@end'</td>
</tr>
<tr>
<td>1030</td>
<td>Conditional compilation turned off</td>
</tr>
<tr>
<td>1031</td>
<td>Expected constant</td>
</tr>
<tr>
<td>1032</td>
<td>Expected '@'</td>
</tr>
<tr>
<td>1033</td>
<td>Expected 'catch'</td>
</tr>
<tr>
<td>1034</td>
<td>Expected 'var'</td>
</tr>
<tr>
<td>1035</td>
<td>'throw' must be followed by an expression on the same source line</td>
</tr>
</tbody>
</table>
## isFinite Method

### See Also

### Applies To

### Description

Returns a Boolean value that indicates if a supplied number is finite.

### Syntax

```
isFinite(number)
```

The `number` argument is a required numeric value.

### Remarks

The `isFinite` method returns `true` if `number` is any value other than `NaN`, negative infinity, or positive infinity. In those three cases, it returns `false`.

---

**Microsoft® JScript®**

**isFinite Method**

**Description**

Returns a Boolean value that indicates if a supplied number is finite.

**Syntax**

```
isFinite(number)
```

The `number` argument is a required numeric value.

**Remarks**

The `isFinite` method returns `true` if `number` is any value other than `NaN`, negative infinity, or positive infinity. In those three cases, it returns `false`.

---

**Language Reference**

**Version 3**
**Description**

Returns the position of the first substring match in a regular expression search.

**Syntax**

```javascript
stringObj.search(rgexp)
```

The **search** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stringObj</td>
<td>Required. The <em>String</em> object or literal to search.</td>
</tr>
<tr>
<td>rgexp</td>
<td>Required. A <em>Regular Expression</em> object containing the pattern to search for.</td>
</tr>
</tbody>
</table>

**Remarks**

The **search** method indicates if a match is present or not. If a match is found, the **search** method returns an integer value that indicates the offset from the beginning of the string where the match occurred. If no match is found, it returns -1. To get further information, use the **match** method.

The following example illustrates the use of the **search** method:
function SearchDemo()
{
    var r, re;
    var s = "The quick brown fox jumped over the");
    r = s.search(re);
    return(r);
}
See Also

Description

Deletes a property from an object, or removes an element from an array.

Syntax

```javascript
delete expression
```

Where `expression` is a valid JScript expression that usually (but does not have to) result in a property name or array element.

Remarks

If the result of `expression` is an object, the property specified in `expression` exists, and the object will not allow it to be deleted, `false` is returned.

In all other cases, `true` is returned.
What Is JScript?

JScript Basics

- Writing JScript Code
- JScript Variables
- JScript Data Types
- JScript Operators
- Controlling Program Flow
- JScript Functions
- JScript Objects
- JScript Reserved Keywords

Advanced JScript

- Recursion
- Variable Scope
- Copying, Passing, and Comparing Data
- Using Arrays
- Advanced Object Creation
- Special Characters
- Troubleshooting Your Scripts

Using JScript In Internet Explorer

- Displaying Information in the Browser
- Using Message Boxes
**ASCII Character Set**
American Standard Code for Information Interchange (ASCII) 7-bit character set widely used to represent letters and symbols found on a standard U.S. keyboard. The ASCII character set is the same as the first 128 characters (0–127) in the ANSI character set.

**Automation object**
An object that is exposed to other applications or programming tools through Automation interfaces.

**bitwise comparison**
A bit-by-bit comparison of identically positioned bits in two numeric expressions.

**Boolean expression**
An expression that evaluates to either **true** or **false**. Non-Boolean expressions are converted to Boolean values, when necessary, according to the following rules:

- All objects are considered true.
- Strings are considered false if and only if they are empty.
- **null** and **undefined** are considered false.
- Numbers are considered false if and only if they are zero.
**character code**
A number that represents a particular character in a set, such as the ASCII character set.

---

**class**
The formal definition of an object. The class acts as the template from which an instance of an object is created at run time. The class defines the properties of the object and the methods used to control the object's behavior.

---

**comment**
Text added to code by a programmer that explains how the code works. In JScript, a comment line generally starts with //. Use the /* and */ delimiters to create a multiline comment.

---

**comparison operator**
A character or symbol indicating a relationship between two or more values or expressions. These operators include less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=), not equal (!=), and equal (==).

---

**compound statement**
A sequence of statements enclosed in braces ({ }). Can be used to perform multiple tasks any time a single statement is expected.
**constructor**

A JavaScript function that has two special features:

- It is invoked by the `new` operator.
- It is passed the address of a newly created object through the `this` keyword.

Use constructors to initialize new objects.

---

**expression**

A combination of keywords, operators, variables, and literals that yield a string, number, or object. An expression can perform a calculation, manipulate characters, call a function, or test data.

---

**intrinsic object**

An object that is part of the standard JavaScript language. These objects are available to all scripts. The intrinsic objects in JavaScript are **Array**, **Boolean**, **Date**, **Function**, **Global**, **Math**, **Number**, **Object**, **RegExp**, **Regular Expression**, and **String**.

---

**local time**

The time on a computer, either a client or server, from where a script is executed.

---

**locale**

The set of information that corresponds to a given language and country. A locale affects the language of predefined programming terms and locale-specific settings. There are two contexts where locale information is
important:

- The code locale affects the language of terms such as keywords and defines locale-specific settings such as the decimal and list separators, date formats, and character sorting order.

- The system locale affects the way locale-aware functionality behaves, for example, when you display numbers or convert strings to dates. You set the system locale using the Control Panel utilities provided by the operating system.

null

A value indicating that a variable contains no valid data. null is the result of:

- An explicit assignment of null to a variable.

- Any operation between expressions that contain null.

numeric expression

Any expression that can be evaluated as a number. Elements of the expression can include any combination of keywords, variables, literals, and operators that result in a number. In certain circumstances, strings are also converted to numbers if possible.

primitive

A data type that is part of the JScript language and manipulated by value. The data types in JScript considered to be primitive are number, Boolean, string, and function. Objects and arrays are not primitive data types.
**property**
A named attribute of an object. Properties define object characteristics such as size, color, and screen location, or the state of an object, such as enabled or disabled.

**run-time error**
An error that occurs when code is running. A run-time error results when a statement attempts an invalid operation.

**scope**
Defines the visibility of a variable, procedure, or object. Variables declared in functions are visible only within the function and lose their value between calls.

**string comparison**
A comparison of two sequences of characters. Unless specified in the function making the comparison, all string comparisons are binary. In English, binary comparisons are case-sensitive; text comparisons are not.

**string expression**
Any expression that evaluates to a sequence of continuous characters. Elements of a string expression can include a function that returns a string, a string literal, a **String** object, or a string variable.

**undefined**
A special value given to variables after they are created and before a value
has been assigned to them.

**Universal Coordinated Time (UTC)**
Universal Coordinated Time, which refers to the time as set by the World Time Standard. Previously referred to as Greenwich Mean time or GMT.

**user-defined object**
An object is one that is created by a user in source code.

**variable**
A location used for storing and manipulating values by name. As JScript is loosely typed, a single variable can hold different types of data over the course of a script.

**wrapper**
An object that is created to provide an object-style interface to some other type of data. The **Number** and **Boolean** objects are examples of wrapper objects.
Microsoft JScript provides nine intrinsic (or "built-in") objects. They are the **Array**, **Boolean**, **Date**, **Function**, **Global**, **Math**, **Number**, **Object**, and **String** objects. Each of the intrinsic objects has associated methods and properties that are described in detail in the [language reference](#). Certain of the objects are also described here.

**Array Object**

In JScript, objects are handled as arrays and arrays are handled as objects. The subscripts of an array, which are entirely equivalent to the properties of an object, can be referred to by number (or by name, if you assign names to them). To create a new array, use the **new** operator and the **Array()** constructor, as in the following example.

```javascript
var theMonths = new Array(12) { 
    theMonths[0] = "Jan";
    theMonths[1] = "Feb";
    theMonths[2] = "Mar";
    theMonths[3] = "Apr";
    theMonths[4] = "May";
    theMonths[5] = "Jun";
    theMonths[6] = "Jul";
    theMonths[7] = "Aug";
    theMonths[8] = "Sep";
};
```
theMonths[9] = "Oct";
theMonths[10] = "Nov";
theMonths[11] = "Dec";
}

When you create an array by using the **Array** keyword, JScript includes in the array a write-only **length** property, which records the number of entries in the array. If you do not specify a number, the length is set to 0, and the array has no entries. If you specify a number, the length is set to that number. If you specify more than one parameter, the parameters are used as entries in the array, and the number of parameters is assigned to the length property, as in the following example, which is equivalent to the preceding one.

```javascript
```

JScript automatically changes the value of **length** if you add elements to an array that you created with the **Array** keyword.

**String Object**

In JScript, strings are objects. This means that any time you declare a string variable or use a string literal, what you're actually doing is creating a new string object. The **String** object has certain built-in methods, which you can use with your strings. One of these is the **substring** method, which returns part of the string. It takes two numbers as its arguments.

```javascript
aString = "0123456789";
var aChunk = aString.substring(4, 7); // Sets aChunk to "456".
```
var aNotherChunk = aString.substring(7, 4); // Sets a

// Using the preceding Array creation example:
firstLetter = theMonths [5].substring(0,1); // Sets the :

Another property of the String object is the length property. This property contains the number of characters in the string, which is 0 for an empty string. This a numeric value, and can be used directly in calculations.

var howLong = "Hello World".length // Sets the howI

Math Object

The Math object has a number of properties and methods, all predefined. The properties are specific numbers. One of these is the value of pi (approximately 3.14159...). This is the Math.PI property, shown in the following example.

// A radius variable is declared and assigned a numeric
var circleArea = Math.PI * radius * radius; // Note ca

One of the built-in methods of the Math object is the exponentiation method, or pow, which raises a number to a specified power. The following example makes use of both pi and exponentiation.

// This formula calculates the volume of a sphere with
volume = (4/3)*(Math.PI*Math.pow(radius,3));

Date Object
Use the **Date** object to capture today's date, and to calculate differences between dates. It has a number of properties and methods, all predefined. In general, the **Date** object provides the day of the week; the month, day, and year; and the time in hours, minutes, and seconds. This information is based on the number of milliseconds since January 1, 1970, 00:00:00.000 GMT. GMT stands for "Greenwich Mean Time"; the preferred term is UTC, or "Universal Coordinated Time," which refers to signals issued by the World Time Standard.

Note As far as JScript is concerned, time begins at midnight on January 1, 1970; you cannot ask JScript to create a **Date** object that represents an earlier time than that. If you need to deal with earlier times you must write your own code to do so, a formidable task.

To create a new **Date** object you use the **new** operator. The following example calculates, for the current year, the number of days that have passed and the number of days that are left.

```javascript
/*

This example uses the array of month names defined |

The first statement assigns today's date, in "Day Month |
format, to the thisIsToday variable.
*/

var thisIsToday = new Date();

var toDay = new Date(); // Capture today's date.

// Extract the year, the month, and the day.
```javascript
var thisYear = toDay.getFullYear() + 1900;
var thisMonth = theMonths[toDay.getMonth()];
var thisDay = thisMonth + " " + toDay.getDate() + ",

// Determine the # of days since the start.
thisDay = Math.round(Date.parse(thisDay)/8.64e7);

// Do the same for the beginning of the year.
var firstDay = "Jan 1, " + thisYear;
firstDay = Math.floor(Date.parse(firstDay)/8.64e7);

// Do it again for the end of the year, in case it's a leap
var lastDay = "Dec 31, " + thisYear;
lastDay = Math.floor(Date.parse(lastDay)/8.64e7);

// Compute the number of days in the year.
var daysInYear = (lastDay - firstDay) + 1;

// Determine how many days have elapsed, and how nr
var daysElapsed = thisDay - firstDay;
var daysLeft = daysInYear - daysElapsed;

// Set up comments for most of the year.
var comment1 = daysElapsed + " days have elapsed in 
var comment2 = "That means there are " + daysLeft + 

// Cover the special cases: beginning & end of year, ar
```

if (daysElapsed == 0) {
  comment1 = "It's January first, " + thisYear + ".";
}
if (daysElapsed == 1) {
  comment1 = "Only one day gone so far.";
}
if (daysElapsed == daysInYear) {
  comment1 = thisYear + " is just about over.";
}

if (daysLeft == 0) {
  comment2 = "Best wishes for the New Year!’’;
}
if (daysLeft == 1) {
  comment2 = "There’s only one day left in " + thisYear
}
if (daysLeft == daysInYear) {
  comment2 = "Happy New Year!";
}

**Number Object**

In addition to the special numeric properties (**PI**, for example) that are available in the **Math** object, several other properties are available in Microsoft JScript through the **Number** object.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Largest possible number, about</td>
</tr>
<tr>
<td><strong>MAX_VALUE</strong></td>
<td>1.79E+308; can be positive or negative. (Value varies slightly from system to system.)</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>MIN_VALUE</strong></td>
<td>Smallest possible number, about 2.22E-308; can be positive or negative. (Value varies slightly from system to system.)</td>
</tr>
<tr>
<td><strong>NaN</strong></td>
<td>Special nonnumeric value, &quot;not a number.&quot;</td>
</tr>
<tr>
<td><strong>POSITIVE_INFINITY</strong></td>
<td>Any positive value larger than Number.MAX_VALUE is automatically converted to this value; represented as &quot;Inf&quot;.</td>
</tr>
<tr>
<td><strong>NEGATIVE_INFINITY</strong></td>
<td>Any negative value larger than -Number.MAX_VALUE is automatically converted to this value; represented as &quot;-Inf&quot;.</td>
</tr>
</tbody>
</table>

**Number.NaN** is a special property that is defined as "not a number."
Division by zero, for example, returns NaN. An attempt to parse a string that cannot be parsed as a number also returns Number.NaN. NaN compares unequal to any number and also to itself. To test for a NaN result, do not compare against Number.NaN; use the isNaN() function instead.
To create instances of an object, you must first define it by giving it properties and, if appropriate, methods. For instance, the following example defines a pasta object. Notice the keyword this, which you use to refer to the current object.

```javascript
function pasta( grain, grain2, width, shape, shapenum, 
{ 
    this.length = 7; // Number of properties in the object 
    this.grain = grain; // What grain is it made of? (string) 
    this.grain2 = grain2; // Any other flour in it? (string) 
    this.width = width; // How wide is it? (number) 
    this.shape = shape; // What is the cross-section? (string) 
    this.shapenum = shapenum; // Is it one of the registered shapes? (number) 
    this.extent = extent; // How long is it? (number) 
    this.egg = egg; // Does it have egg yolk as a binder? 
}
```

Once you define an object, you create instances of it with the new operator.

```javascript
var spaghetti = new pasta("wheat", "", 0.2, "circle", 9,
var linguine = new pasta("wheat", "", 0.3, "oval", 17,
```

You can add properties to one instance of an object, to change that instance, but those properties do not become part of the definition of the object, and do not show up in other instances
unless you specifically add them. If you want the extra properties to show up in all instances of the object, you must add them to the object definition.

    // Additional properties for spaghetti.
    spaghetti.color = "pale straw";
    spaghetti.drycook = 7;
    spaghetti.freshcook = 0.5;
    
    var chowFun = new pasta("rice", "", 3, "flat", , 12, false);

    // Neither the chowFun object, the linguine object, nor the pasta
    // object definition has the three extra properties given to the spaghetti object.

Including Methods in the Definition

It is possible to include methods in the definition of an object. The following example builds an object that consists of an array of strings, and a method. The method adds a string to the array, increasing its size in order to do so. Notice that this makes each instance of the object indefinitely extensible.

    function addItem(newItem) // Define a function to extend the list.
    {
        this.length += 1; // Increment the length of the array.
        this[(this.length-1)] = newItem; // Add the new item, maintaining item numbering.
    }
function shoppingList(firstItem) // Define a "shopping
{
    this.length = 2; // Number of properties in the object
    this.addItem = addItem; // Include the addItem function
    this[(this.length-1)] = firstItem; // The first item is 1
}

var myList = new shoppingList("Milk");
myList.addItem("Eggs"); // Use the method to add Eggs
myList.addItem("Breadfruit"); // Breadfruit becomes

At this point, the contents of the array are as follows:

- myList[length] is 4
- myList[addItem] is the addItem function
- myList[1] is Milk
- myList[2] is Eggs
- myList[3] is Breadfruit

Note that the indexing is not exactly as you might expect it to be if it were handled in a strictly numeric way. If you execute a for...in loop on this array, the loop iterates in the order given here, and the loop variable has the initial value "length" rather than 0.
Using alert, prompt, and confirm

Use alert, confirm, and prompt message boxes to obtain input from your user. The boxes are methods of the interface `window` object. Because the `window` object is at the top of the object hierarchy, you do not actually have to use the full name (for example, "window.alert()") of any of these message boxes, but it is a good idea to do so, because it helps you remember to which object they belong.

Alert Message Box

The `alert` method has one argument, the string of text you want to display to the user. The string is not HTML. The message box provides an OK button so the user can close it and is modal, that is, the user must close the message box before continuing.

```
window.alert("Welcome! Press OK to continue.
```

Confirm Message Box

The confirm message box lets you ask the user a "yes-or-no" question, and gives the user the option of clicking either an OK button or a Cancel button. The `confirm` method returns either `true` or `false`. This message box is also modal: the user must respond to it (click a button), and thereby close it, before proceeding.
var truthBeTold = window.confirm("Click OK to continue.
Click Cancel to stop.");
if (truthBeTold) {
    window.alert("Welcome to our Web page!");
} else window.alert("Bye for now!");

Prompt Message Box

The prompt message box provides a text field in which the user can type an answer in response to your prompt. This box has an OK button and a Cancel button. If you provide a second string argument, the prompt message box displays that second string in the text field, as the default response. Otherwise, the default text is ":".

Like the alert() and confirm() methods, prompt displays a modal message box. The user must close it before continuing.

var theResponse = window.prompt("Welcome?", "Enter your name here.");
Welcome to the JScript Language Reference

These handy blocks of information will help you explore the many different parts of JScript.

You'll find all the parts of the JScript language listed alphabetically under the Alphabetic Keyword List. But if you want to examine just one category, say, objects, each language category has its own, more compact section.

How's it work? Click on one of the headings to the left to display a list of items contained in that category. From this list, select the topic that you want to view. Once you've opened that topic, you can easily link to other related sections.

So, go ahead and take a look! Study some statements, mull over the methods, or figure out a few functions. You'll see just how versatile the JScript language can be!
| **Microsoft Scripting Run-time Features** | List of scripting run-time features currently in JScript. |

© 2000 Microsoft Corporation. All rights reserved.
Add Method
(Dictionary)

Description

Adds a key and item pair to a Dictionary object.

Syntax

object.Add (key, item)

The Add method has the following parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a Dictionary object.</td>
</tr>
<tr>
<td>key</td>
<td>Required. The key associated with the item being added.</td>
</tr>
<tr>
<td>item</td>
<td>Required. The item associated with the key being added.</td>
</tr>
</tbody>
</table>

Remarks

An error occurs if the key already exists.

The following example illustrates the use of the Add method:

```javascript
var d;
d = new ActiveXObject("Scripting.Dictionary");
d.Add("a", "Athens");
```
d.Add("b", "Belgrade");
d.Add("c", "Cairo");
Add Method (Folders)

Description

Adds a new **Folder** to a **Folders** collection.

Syntax

```javascript
object.Add(folderName)
```

The **Add** method has the following parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <strong>Folders</strong> collection.</td>
</tr>
<tr>
<td><code>folderName</code></td>
<td>Required. The name of the new <strong>Folder</strong> being added.</td>
</tr>
</tbody>
</table>

Remarks

The following example illustrates the use of the **Add** method to create a new folder:

```javascript
function AddNewFolder(path,folderName) {
    var fso, f, fc, nf;
    fso = new ActiveXObject("Scripting.FileSystemObject");
```
f = fso.GetFolder(path);
fC = f.SubFolders;
if (folderName != "")
    nf = fc.Add(folderName);
else
    nf = fc.Add("New Folder");
}

An error occurs if the folderName already exists.
**Microsoft® JScript®**

**AtEndOfLine Property**

**See Also**

**Applies To**

---

**Description**

Returns `true` if the file pointer is positioned immediately before the end-of-line marker in a TextStream file; `false` if it is not. Read-only.

**Syntax**

```
object.AtEndOfLine
```

The `object` is always the name of a TextStream object.

**Remarks**

The AtEndOfLine property applies only to TextStream files that are open for reading; otherwise, an error occurs.

The following code illustrates the use of the AtEndOfLine property:

```javascript
function GetALine(filespec)
{
    var fso, a, s, ForReading;
    ForReading = 1, s = "";
    fso = new ActiveXObject("Scripting.FileSystemObject");
```
a = fso.OpenTextFile(filespec, ForReading, false);
while (!a.AtEndOfLine)
{
    s += a.Read(1);
}

a.Close();
return(s);
AtEndOfStream Property

See Also | Applies To

Description

Returns true if the file pointer is at the end of a TextStream file; false if it is not. Read-only.

Syntax

\texttt{object.AtEndOfStream}

The object is always the name of a TextStream object.

Remarks

The AtEndOfStream property applies only to TextStream files that are open for reading, otherwise, an error occurs.

The following code illustrates the use of the AtEndOfStream property:

```
function GetALine(filespec)
{
    var fso, f, s, ForReading;
    ForReading = 1, s = "";
```
fso = new ActiveXObject("Scripting.FileSystemObject");
f = fso.OpenTextFile(filespec, ForReading, false);
while (!f.AtEndOfStream)
    s += f.ReadLine();
f.Close();
return(s);
}
**Attributes Property**

**See Also**  
**Applies To**

---

**Description**

Sets or returns the attributes of files or folders. Read/write or read-only, depending on the attribute.

**Syntax**

```javascript
object.Attributes [ = newattributes ]
```

The `Attributes` property has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <code>File</code> or <code>Folder</code> object.</td>
</tr>
<tr>
<td><code>newattributes</code></td>
<td>Optional. If provided, <code>newattributes</code> is the new value for the attributes of the specified <code>object</code>.</td>
</tr>
</tbody>
</table>

**Settings**

The `newattributes` argument can have any of the following values or any logical combination of the following values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0</td>
<td>Normal file. No attributes are set.</td>
</tr>
<tr>
<td>ReadOnly</td>
<td>1</td>
<td>Read-only file. Attribute is read/write.</td>
</tr>
<tr>
<td>Hidden</td>
<td>2</td>
<td>Hidden file. Attribute is read/write.</td>
</tr>
<tr>
<td>----------</td>
<td>----</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>System</td>
<td>4</td>
<td>System file. Attribute is read/write.</td>
</tr>
<tr>
<td>Volume</td>
<td>8</td>
<td>Disk drive volume label. Attribute is read-only.</td>
</tr>
<tr>
<td>Directory</td>
<td>16</td>
<td>Folder or directory. Attribute is read-only.</td>
</tr>
<tr>
<td>Archive</td>
<td>32</td>
<td>File has changed since last backup. Attribute is read/write.</td>
</tr>
<tr>
<td>Alias</td>
<td>64</td>
<td>Link or shortcut. Attribute is read-only.</td>
</tr>
<tr>
<td>Compressed</td>
<td>128</td>
<td>Compressed file. Attribute is read-only.</td>
</tr>
</tbody>
</table>

**Remarks**

The following code illustrates the use of the **Attributes** property with a file:

```javascript
function ToggleArchiveBit(filespec) {
    var fso, f, r, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    f = fso.GetFile(filespec);
    if (f.attributes && 32) {
        f.attributes = f.attributes - 32;
        s = "Archive bit is cleared.";
    } else {
        f.attributes = f.attributes + 32;
        s = "Archive bit is set.";
    }
    return s;
}
```
return(s);

}
AvailableSpace Property

Description

Returns the amount of space available to a user on the specified drive or network share.

Syntax

object.AvailableSpace

The object is always a Drive object.

Remarks

The 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.

The following code illustrates the use of the AvailableSpace property:

```javascript
function ShowAvailableSpace(drvPath)
{
    var fso, d, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    d = fso.GetFolder(drvPath);
    s = d.AvailableSpace;
    // Use s to display available space information
}
```
d = fso.GetDrive(fso.GetDriveName(drvPath))
s = "Drive " + drvPath.toUpperCase() + " - ";
s += d.VolumeName + "<br>";
s += "Available Space: " + d.AvailableSpace/1
return(s);
Microsoft® JScript® *BuildPath*

**Method**

**See Also**  **Applies To**

---

**Description**

Appends a name to an existing path.

**Syntax**

*object*.BuildPath(*path*, *name*)

The **BuildPath** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>object</em></td>
<td>Required. Always the name of a <strong>FileSystemObject</strong>.</td>
</tr>
<tr>
<td><em>path</em></td>
<td>Required. Existing path to which <em>name</em> is appended. Path can be absolute or relative and need not specify an existing folder.</td>
</tr>
<tr>
<td><em>name</em></td>
<td>Required. Name being appended to the existing <em>path</em>.</td>
</tr>
</tbody>
</table>

**Remarks**

The **BuildPath** method inserts an additional path separator between the existing path and the name, only if necessary.

The following example illustrates use of the **BuildPath** method:

```javascript
function GetBuildPath(path)
```

{  
    var fso, newpath;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    newpath = fso.BuildPath(path, "New Folder");
    return(newpath);
}

See Also

Description

Closes an open TextStream file.

Syntax

```
object.Close();
```

The `object` is always the name of a TextStream object.

Remarks

The following example illustrates use of the Close method:

```
var fso;
fso = new ActiveXObject("Scripting.FileSystemObject");
a = fso.CreateTextFile("c:\testfile.txt", true);
a.WriteLine("This is a test.");
a.Close();
```
Column Property

Description

Read-only property that returns the column number of the current character position in a TextStream file.

Syntax

object.Column

The object is always the name of a TextStream object.

Remarks

After a newline character has been written, but before any other character is written, Column is equal to 1.

The following examples illustrates the use of the Column property:

```javascript
function GetColumn() {
    var fso, f, m;
    var ForReading = 1, ForWriting =
    fso = new ActiveXObject("Scripti
f = fso.OpenTextFile("c:\testfile.txt", ForWriting, true);
f.Write("Hello World!");
f.Close();
f = fso.OpenTextFile("c:\testfile.txt");
m = f.ReadLine();
return(f.Column);
}
**Description**

Copies a specified file or folder from one location to another.

**Syntax**

```javascript
object.Copy( destination[, overwrite] );
```

The `Copy` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <code>File</code> or <code>Folder</code> object.</td>
</tr>
<tr>
<td><code>destination</code></td>
<td>Required. Destination where the file or folder is to be copied. Wildcard characters are not allowed.</td>
</tr>
<tr>
<td><code>overwrite</code></td>
<td>Optional. Boolean value that is <code>True</code> (default) if existing files or folders are to be overwritten; <code>False</code> if they are not.</td>
</tr>
</tbody>
</table>

**Remarks**

The results of the `Copy` method on a `File` or `Folder` are identical to operations performed using `FileSystemObject.CopyFile` or `FileSystemObject.CopyFolder` where the file or folder referred to by `object` is passed as an argument. You should note, however, that the alternative methods are capable of copying multiple files.
or folders.

The following example illustrates the use of the **Copy** method:

```javascript
var fso, f;
  fso = new ActiveXObject("Scripting.FileSystemObject");
  f = fso.CreateTextFile("c:\testfile.txt", true);
  f.WriteLine("This is a test.");
  f.Close();
  f = fso.GetFile("c:\testfile.txt");
  f.Copy("c:\windows\desktop\test2.txt");
```
**CopyFile Method**

**Description**

Copies one or more files from one location to another.

**Syntax**

```javascript
object.CopyFile( source, destination[, overwrite] )
```

The **CopyFile** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. The <em>object</em> is always the name of a <em>FileSystemObject</em>.</td>
</tr>
<tr>
<td>source</td>
<td>Required. Character string file specification, which can include wildcard characters, for one or more files to be copied.</td>
</tr>
<tr>
<td>destination</td>
<td>Required. Character string destination where the file or files from <em>source</em> are to be copied. Wildcard characters are not allowed.</td>
</tr>
<tr>
<td>overwrite</td>
<td>Optional. Boolean value that indicates if existing files are to be overwritten. If <em>true</em>, files are overwritten; if <em>false</em>, they are not. The default is <em>true</em>. Note that <strong>CopyFile</strong> will fail if <em>destination</em> has the read-only attribute set, regardless of the value of <em>overwrite</em>.</td>
</tr>
</tbody>
</table>

**Remarks**
Wildcard characters can only be used in the last path component of the source argument. For example, you can use:

```javascript
fso = new ActiveXObject("Scripting.FileSystemObject");
fso.CopyFile ("c:\mydocuments\letters\*\doc", "c:\t"
```

But you can't use:

```javascript
fso = new ActiveXObject("Scripting.FileSystemObject");
fso.CopyFile ("c:\mydocuments\*\R1??97.xls", "c:
```

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 an individual 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.

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.
**CopyFolder Method**

**See Also**

**Applies To**

**Description**

Recursively copies a folder from one location to another.

**Syntax**

```javascript
object.CopyFolder ( source, destination[, overwrite] );
```

The `CopyFolder` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <code>FileSystemObject</code>.</td>
</tr>
<tr>
<td><code>source</code></td>
<td>Required. Character string folder specification, which can include wildcard characters, for one or more folders to be copied.</td>
</tr>
<tr>
<td><code>destination</code></td>
<td>Required. Character string destination where the folder and subfolders from <code>source</code> are to be copied. Wildcard characters are not allowed.</td>
</tr>
<tr>
<td><code>overwrite</code></td>
<td>Optional. Boolean value that indicates if existing folders are to be overwritten. If <code>true</code>, files are overwritten; if <code>false</code>, they are not. The default is <code>true</code>.</td>
</tr>
</tbody>
</table>

**Remarks**

Wildcard characters can only be used in the last path component
of the \textit{source} argument. For example, you can use:

\begin{verbatim}
fso = new ActiveXObject("Scripting.FileSystemObject"); fso.CopyFolder ("c:\mydocuments\letters\**", "c:\tempfolder")
\end{verbatim}

But you can't use:

\begin{verbatim}
fso = new ActiveXObject("Scripting.FileSystemObject"); fso.CopyFolder ("c:\mydocuments\*\**", "c:\tempfolder")
\end{verbatim}

If \textit{source} contains wildcard characters or \textit{destination} ends with a path separator (\textbackslash{}), it is assumed that \textit{destination} is an existing folder in which to copy matching folders and subfolders. Otherwise, \textit{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 \textit{destination} does not exist, the \textit{source} folder and all its contents gets copied. This is the usual case.

- If \textit{destination} is an existing file, an error occurs.

- If \textit{destination} is a directory, an attempt is made to copy the folder and all its contents. If a file contained in \textit{source} already exists in \textit{destination}, an error occurs if \textit{overwrite} is \texttt{false}. Otherwise, it will attempt to copy the file over the existing file.

- If \textit{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 \textit{overwrite} is \texttt{false}.

An error also occurs if a \textit{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 any changes made before an error occurs.
Count Property

See Also

Applies To

Description

Returns the number of items in a collection or Dictionary object. Read-only.

Syntax

object.Count

The object is always the name of one of the items in the Applies To list.

Remarks

The following code illustrates use of the Count property:

```javascript
function CountDemo()
{
    var a, d, i, s; // Create some variables.
    d = new ActiveXObject("Scripting.Dictionary");
    d.Add("a", "Athens"); // Add some keys and it
    d.Add("b", "Belgrade");
    d.Add("c", "Cairo");
    a = (new VBAArray(d.Keys())); // Get the keys.
    s = "";
```
for (i = 0; i < d.Count; i++)  //Iterate the dictionary.
{
    s += a.getItem(i) + " - " + d(a.getItem(i)) + "<br>";
}
return(s);                   // Return the results.
}
Microsoft® JScript® CreateFolder Method

See Also Applies To

Description

Creates a folder.

Syntax

object.CreateFolder(foldername)

The CreateFolder method has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a FileSystemObject.</td>
</tr>
<tr>
<td>foldername</td>
<td>Required. String expression that identifies the folder to create.</td>
</tr>
</tbody>
</table>

Remarks

An error occurs if the specified folder already exists.

The following code illustrates how to use the CreateFolder method to create a folder:

```javascript
var fso = new ActiveXObject("Scripting.FileSystemObject");
var a = fso.CreateFolder("c:\new folder");
```
Microsoft® JScript® CreateTextFile Method

See Also  Applies To

Description

Creates a specified file name and returns a TextStream object that can be used to read from or write to the file.

Syntax

object.CreateTextFile(filename[, overwrite[, unicode]])

The CreateTextFile method has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a FileSystemObject or Folder object.</td>
</tr>
<tr>
<td>filename</td>
<td>Required. String expression that identifies the file to create.</td>
</tr>
<tr>
<td>overwrite</td>
<td>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.</td>
</tr>
<tr>
<td>unicode</td>
<td>Optional. Boolean value that indicates whether the file is created as a Unicode or ASCII file. The value is true if the file is created as a Unicode file, false if it's created as an ASCII file. If omitted, an ASCII file is assumed.</td>
</tr>
</tbody>
</table>
Remarks

The following code illustrates how to use the `CreateTextFile` method to create and open a text file:

```javascript
var fso = new ActiveXObject("Scripting.FileSystemObject");
var a = fso.CreateTextFile("c:\testfile.txt", true);
a.WriteLine("This is a test.");
a.Close();
```

If the `overwrite` argument is `false`, or is not provided, for a filename that already exists, an error occurs.
Microsoft® JScript® DateCreated Property

See Also

Applies To

Description

Returns the date and time that the specified file or folder was created. Read-only.

Syntax

object.DateCreated

The object is always a File or Folder object.

Remarks

The following code illustrates the use of the DateCreated property with a file:

```javascript
function ShowFileInfo(filespec)
{
    var fso, f, s;
fso = new ActiveXObject("Scripting.FileSystemObject");
f = fso.GetFile(filespec);
s = "Created: " + f.DateCreated;
return(s);
}
```
**DateLastAccessed Property**

**Description**

Returns the date and time that the specified file or folder was last accessed. Read-only.

**Syntax**

```javascript
object.DateLastAccessed
```

The `object` is always a `File` or `Folder` object.

**Remarks**

The following code illustrates the use of the `DateLastAccessed` property with a file:

```javascript
function ShowFileAccessInfo(filespec)
{
    var fso, f, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    f = fso.GetFile(filespec);
    s = filespec.toUpperCase() + "<br>";
    s += "Created: " + f.DateCreated + "<br>";
```
Important This method depends on the underlying operating system for its behavior. If the operating system does not support providing time information, none will be returned.
DateLastModified Property

See Also          Applies To

Description

Returns the date and time that the specified file or folder was last modified. Read-only.

Syntax

object.DateLastModified

The object is always a File or Folder object.

Remarks

The following code illustrates the use of the DateLastModified property with a file:

```javascript
function ShowFileAccessInfo(filespec) {
    var fso, f, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    f = fso.GetFile(filespec);
    s = filespec.toUpperCase() + "<br>";
    s += "Created: " + f.DateCreated + "<br>";
```
s += "Last Accessed: " + f.DateLastAccessed + "<br>
" + f.DateLastModified;
return(s);
}
Delete Method

Description

Deletes a specified file or folder.

Syntax

object.Delete( force );

The **Delete** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a <strong>File</strong> or <strong>Folder</strong> object.</td>
</tr>
<tr>
<td>force</td>
<td>Optional. Boolean value that is <strong>True</strong> if files or folders with the read-only attribute set are to be deleted; <strong>False</strong> (default) if they are not.</td>
</tr>
</tbody>
</table>

Remarks

An error occurs if the specified file or folder does not exist.

The results of the **Delete** method on a **File** or **Folder** are identical to operations performed using `FileSystemObject.DeleteFile` or `FileSystemObject.DeleteFolder`.

The **Delete** 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.
The following example illustrates the use of the `Delete` method:

```javascript
var fso, f;
fso = new ActiveXObject("Scripting.FileSystemObject");
f = fso.CreateTextFile("c:\testfile.txt", true);
f.WriteLine("This is a test.");
f.Close();
f = fso.GetFile("c:\testfile.txt");
f.Delete();
```
Microsoft® JScript® **DeleteFile** Method

**See Also**

**Applies To**

---

**Description**

Deletes a specified file.

**Syntax**

```javascript
object.DeleteFile ( filespec[, force] );
```

The **DeleteFile** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <code>FileSystemObject</code>.</td>
</tr>
<tr>
<td><code>filespec</code></td>
<td>Required. The name of the file to delete. The <code>filespec</code> can contain wildcard characters in the last path component.</td>
</tr>
<tr>
<td><code>force</code></td>
<td>Optional. Boolean value that is <code>true</code> if files with the read-only attribute set are to be deleted; <code>false</code> (default) if they are not.</td>
</tr>
</tbody>
</table>

**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.
The following example illustrates the use of the `DeleteFile` method:

```javascript
function DeleteFile(filespec)
{
    var fso;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    fso.DeleteFile(filespec);
}
```
**DeleteFolder Method**

**See Also**

**Applies To**

---

**Description**

Deletes a specified folder and its contents.

**Syntax**

```javascript
object.DeleteFolder ( folderspec[, force] );
```

The `DeleteFolder` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <code>FileSystemObject</code>.</td>
</tr>
<tr>
<td><code>folderspec</code></td>
<td>Required. The name of the folder to delete. The <code>folderspec</code> can contain wildcard characters in the last path component.</td>
</tr>
<tr>
<td><code>force</code></td>
<td>Optional. Boolean value that is <code>true</code> if folders with the read-only attribute set are to be deleted; <code>false</code> (default) if they are not.</td>
</tr>
</tbody>
</table>

**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.

The following example illustrates the use of the **DeleteFolder** method:

```javascript
function DeleteFolder(folderspec)
{
    var fso;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    fso.DeleteFolder(folderspec);
}
```
**Dictionary Object**

**See Also**  
**Methods**  
**Properties**

**Description**

Object that stores data key, item pairs.

**Syntax**

```javascript
y = new ActiveXObject("Scripting.Dictionary")
```

**Remarks**

A **Dictionary** object is the equivalent of a PERL associative array. Items can be any form of data, and are stored in the array. Each item is associated with a unique key. The key is used to retrieve an individual item and is usually a integer or a string, but can be anything except an array.

The following code illustrates how to create a **Dictionary** object:

```javascript
var y = new ActiveXObject("Scripting.Dictionary");
y.add ("a", "test");
if (y.Exists("a"))
    document.write("true");
...
```
**Drive Object**

**Description**

Provides access to the properties of a particular disk drive or network share.

**Remarks**

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

```javascript
function ShowFreeSpace(drvPath)
{
    var fso, d, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    d = fso.GetDrive(fso.GetDriveName(drvPath));
    s = "Drive " + drvPath + " - ";
    s += d.VolumeName + "\n";
    s += "Free Space: " + d.FreeSpace/1024 + " Kbytes";
    return(s);
}
```
**Description**

Returns the drive letter of the drive on which the specified file or folder resides. Read-only.

**Syntax**

```
object.Drive
```

The `object` is always a **File** or **Folder** object.

**Remarks**

The following code illustrates the use of the **Drive** property:

```javascript
function ShowFileAccessInfo(filespec) {
    var fso, f, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    f = fso.GetFile(filespec);
    s = f.Name + " on Drive " + f.Drive + "<br>");
    s += "Created: " + f.DateCreated + "<br>");
    s += "Last Accessed: " + f.DateLastAccessed + "<br>");
    s += "Last Modified: " + f.DateLastModified;
```
return(s);
}

Microsoft® JScript® DriveExists Method

See Also

Applies To

Description

Returns True if the specified drive exists; False if it does not.

Syntax

object.DriveExists(drivespec)

The DriveExists method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a FileSystemObject.</td>
</tr>
<tr>
<td>drivespec</td>
<td>Required. A drive letter or a complete path specification.</td>
</tr>
</tbody>
</table>

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.

The following example illustrates the use of the DriveExists method:

```javascript
function ReportDriveStatus(drv)
{
```
var fso, s = "";
fso = new ActiveXObject("Scripting.FileSystemObject");
if (fso.DriveExists(drv))
    s += "Drive " + drv + " exists."
else
    s += "Drive " + drv + " doesn't exist.";
return(s);
}
Microsoft® JScript® DriveLetter Property

Description

Returns the drive letter of a physical local drive or a network share. Read-only.

Syntax

```
object.DriveLetter
```

The object is always a Drive object.

Remarks

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.

The following code illustrates the use of the DriveLetter property:

```javascript
function ShowDriveLetter(drvPath)
{
    var fso, d, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    d = fso.GetDrive(fso.GetDriveName(drvPath));
    console.log(d.DriveLetter);
}
```
s = "Drive " + d.DriveLetter.toUpperCase() + 
     s += d.VolumeName + "<br>
     s += "Available Space: " + d.AvailableSpace/1
            return(s);
Drives Collection

Description

Read-only collection of all available drives.

Remarks

Removable-media drives need not have media inserted for them to appear in the Drives collection.

The following example illustrates how to get the Drives collection using the Drives property and iterate the collection using the Enumerator object:

```javascript
function ShowDriveList()
{
    var fso, s, n, e, x;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    e = new Enumerator(fso.Drives);
    s = ";";
    for (; !e.atEnd(); e.moveNext())
    {
        x = e.item();
        s = s + x.DriveLetter;
    }
}```
s += " - ";
if (x.DriveType == 3)
    n = x.ShareName;
elser if (x.IsReady)
    n = x.VolumeName;
else
    n = "[Drive not ready]";
s += n + "<br>");
}
return(s);
}
**Description**

Returns a **Drives** collection consisting of all **Drive** objects available on the local machine.

**Syntax**

```
object.Drives
```

The *object* is always a **FileSystemObject**.

**Remarks**

Removable-media drives need not have media inserted for them to appear in the **Drives** collection.

You can iterate the members of the **Drives** collection using the **Enumerator** object and the **for** statement:

```
function ShowDriveList()
{
    var fso, s, n, e, x;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    e = new Enumerator(fso.Drives);
```
s = "";
for (; !e.atEnd(); e.moveNext())
{
    x = e.item();
    s = s + x.DriveLetter;
    s += " - ";
    if (x.DriveType == 3)
        n = x.ShareName;
    else if (x.IsReady)
        n = x.VolumeName;
    else
        n = "[Drive not ready]";
    s += n + "<br>";
}
return(s);
**DriveType Property**

**See Also**

**Applies To**

---

**Description**

Returns a value indicating the type of a specified drive.

**Syntax**

```javascript
object.DriveType
```

The `object` is always a `Drive` object.

**Remarks**

The following code illustrates the use of the `DriveType` property:

```javascript
function ShowDriveType(drvpath)
{
    var fso, d, s, t;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    d = fso.GetDrive(drvpath);
    switch (d.DriveType)
    {
        case 0: t = "Unknown"; break;
        case 1: t = "Removable"; break;
    }
}
```
case 2: t = "Fixed"; break;
case 3: t = "Network"; break;
case 4: t = "CD-ROM"; break;
case 5: t = "RAM Disk"; break;
}
s = "Drive " + d.DriveLetter + ": - " + t;
return(s);
}
**Description**

Returns **true** if a specified key exists in the **Dictionary** object, **false** if it does not.

**Syntax**

\[ \text{object.Exists(key)} \]

The **Exists** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a <strong>Dictionary</strong> object.</td>
</tr>
<tr>
<td>key</td>
<td>Required. Key value being searched for in the <strong>Dictionary</strong> object.</td>
</tr>
</tbody>
</table>

The following example illustrates the use of the **Exists** method:

```javascript
function keyExists(k)
{
    var fso, s = "";
    d = new ActiveXObject("Scripting.Dictionary");
    d.Add("a", "Athens");
    d.Add("b", "Belgrade");
```
d.Add("c", "Cairo");
if (d.Exists(k))
    s += "Specified key exists."
else
    s += "Specified key doesn't exist."
return(s);
See Also

Properties

Methods

Description

Provides access to all the properties of a file.

Remarks

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

```javascript
function ShowFileInfo(filespec)
{
    var fso, f, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
f = fso.GetFile(filespec);
s = f.DateCreated;
return(s);
}
```
**FileExists Method**

**See Also**  
**Applies To**

---

**Description**

Returns **True** if a specified file exists; **False** if it does not.

**Syntax**

```
object.FileExists(filespec)
```

The `FileExists` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <code>FileSystemObject</code>.</td>
</tr>
<tr>
<td><code>filespec</code></td>
<td>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.</td>
</tr>
</tbody>
</table>

The following example illustrates the use of the `FileExists` method:

```javascript
function ReportFileStatus(filespec)
{
    var fso, s =.filespec;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    if (fso.FileExists(filespec))
```
s += " exists.";
else
  s += " doesn't exist.";
return(s);
}
Files Collection

Description

Collection of all File objects within a folder.

Remarks

The following example illustrates how to get a Files collection and iterate the collection using the Enumerator object and the for statement:

```javascript
function ShowFolderFileList(folderspec)
{
    var fso, f, f1, fc, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    f = fso.GetFolder(folderspec);
    fc = new Enumerator(f.files);
    s = "";
    for (; !fc.atEnd(); fc.moveNext())
    {
        s += fc.item();
        s += "<br>";
    }
    return(s);
}
```
Description

Returns a **Files** collection consisting of all **File** objects contained in the specified folder, including those with hidden and system file attributes set.

Syntax

```
object.Files
```

The *object* is always a **Folder** object.

Remarks

The following code illustrates the use of the **Files** property:

```
function ShowFolderFileList(folderspec) {
    var fso, f, fc, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    f = fso.GetFolder(folderspec);
    fc = new Enumerator(f.files);
    s = "";
    for (; !fc.atEnd(); fc.moveNext())
```
{  
s += fc.item();  
s += "<br>";  
}  
return(s);  
}
**FileSyncObject**

**Description**

Provides access to a computer's file system.

**Syntax**

```javascript
y = new ActiveXObject("Scripting.FileSystemObject")
```

**Remarks**

The following code illustrates how the **FileSystemObject** is used to return a **TextStream** object that can be read from or written to:

```javascript
var fso = new ActiveXObject("Scripting.FileSystemObject")
var a = fso.CreateTextFile("c:\testfile.txt", true);
a.WriteLine("This is a test.");
a.Close();
```

In the example code, the **ActiveXObject** object is assigned to the **FileSystemObject** (fso). The **CreateTextFile** method then creates the file as a **TextStream** object (a), and the **WriteLine** method writes a line of text to the created text file. The **Close** method flushes the buffer and closes the file.
**FileSystem Property**

**Description**

Returns the type of file system in use for the specified drive.

**Syntax**

```javascript
object.FileSystem
```

The `object` is always a `Drive` object.

**Remarks**

Available return types include FAT, NTFS, and CDFS.

The following code illustrates the use of the `FileSystem` property:

```javascript
function ShowFileSystemType(drvPath) {
    var fso,d, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    d = fso.GetDrive(drvPath);
    s = d.FileSystem;
    return(s);
}
```
Folder Object

Description

Provides access to all the properties of a folder.

Remarks

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

```javascript
function ShowFolderInfo(folderspec)
{
    var fso, folder, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    folder = fso.GetFolder(folderspec);
    s = folder.DateCreated;
    return(s);
}
```
Microsoft® JScript® Folders Collection

Description

Collection of all Folder objects contained within a Folder object.

Remarks

The following example illustrates how to get a Folders collection and how to iterate the collection using the Enumerator object and the for statement:

```javascript
function ShowFolderList(folderspec) {
    var fso, f, fc, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    f = fso.GetFolder(folderspec);
    fc = new Enumerator(f.SubFolders);
    s = ";
    for (; !fc.atEnd(); fc.moveNext()) {
        s += fc.item();
        s += ";
    }
}
```
return(s);
}

FolderExists Method

Description

Returns **True** if a specified folder exists; **False** if it does not.

Syntax

\[ \text{object.FolderExists(folderspec)} \]

The `FolderExists` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <code>FileSystemObject</code>.</td>
</tr>
<tr>
<td><code>folderspec</code></td>
<td>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.</td>
</tr>
</tbody>
</table>

The following example illustrates the use of the `FolderExists` method:

```javascript
function ReportFolderStatus(fldr) {
    var fso, s = fldr;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    if (fso.FolderExists(fldr))
```
s += " exists."

else
    s += " doesn't exist."

return(s);
Description

Returns the amount of free space available to a user on the specified drive or network share. Read-only.

Syntax

object.FreeSpace

The object is always a Drive object.

Remarks

The 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.

The following code illustrates the use of the FreeSpace property:

```javascript
function ShowFreeSpace(drvPath)
{
    var fso, d, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    d = driveObject(drvPath);
    s = fso.FreeSpace(d);
    alert(s);
}
```
d = fso.GetDrive(fso.GetDriveName(drvPath))
s = "Drive " + drvPath.toUpperCase() + " - ";
s += d.VolumeName + "<br>"
s += "Free Space: " + d.FreeSpace/1024 + " K"
return(s);
Description

Returns a complete and unambiguous path from a provided path specification.

Syntax

object.GetAbsolutePathName(pathspec)

The GetAbsolutePathName method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a FileSystemObject.</td>
</tr>
<tr>
<td>pathspec</td>
<td>Required. Path specification to change to a complete and unambiguous path.</td>
</tr>
</tbody>
</table>

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.
The following example illustrates the use of the `GetAbsolutePathName` method:

```javascript
function ShowAbsolutePath(path)
{
    var fso, s = "";
    fso = new ActiveXObject("Scripting.FileSystemObject");
    s += fso.GetAbsolutePathName(path);
    return(s);
}
```
GetBaseName Method

Description

Returns a string containing the base name of the last component, less any file extension, in a path.

Syntax

Object.GetBaseName(path)

The GetBaseName method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a FileSystemObject.</td>
</tr>
<tr>
<td>path</td>
<td>Required. The path specification for the component whose base name is to be returned.</td>
</tr>
</tbody>
</table>

Remarks

The GetBaseName method returns a zero-length string (""") if no component matches the path argument.

Note 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 following example illustrates the use of the **GetBaseName** method:

```javascript
function ShowBaseName(filespec) {
    var fso, s = "";
    fso = new ActiveXObject("Scripting.FileSystemObject");
    s += fso.GetBaseName(filespec);
    return(s);
}
```
Description

Returns a **Drive** object corresponding to the drive in a specified path.

Syntax

\[ \text{object}\text{.GetDrive ( drivespec );} \]

The **GetDrive** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>object</strong></td>
<td>Required. Always the name of a <strong>FileSystemObject</strong>.</td>
</tr>
<tr>
<td><strong>drivespec</strong></td>
<td>Required. The <strong>drivespec</strong> 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).</td>
</tr>
</tbody>
</table>

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.
To call the **GetDrive** method on a normal path string, use the following sequence to get a string that is suitable for use as `drivespec`:

```
DriveSpec = GetDriveName(GetAbsolutePathName(Path))
```

The following example illustrates the use of the **GetDrive** method:

```javascript
function ShowFreeSpace(drvPath)
{
    var fso, d, s ="";
    fso = new ActiveXObject("Scripting.FileSystemObject");
    d = fso.GetDrive(fso.GetDriveName(drvPath));
    s = "Drive " + drvPath.toUpperCase() + " - ";
    s += d.VolumeName + "<br>
    s += "Free Space: " + d.FreeSpace/1024 + " K"
    return(s);
}
```
**GetDriveName Method**

**See Also**

**Applies To**

---

**Description**

Returns a string containing the name of the drive for a specified path.

**Syntax**

```javascript
object.GetDriveName(path)
```

The `GetDriveName` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <code>FileSystemObject</code>.</td>
</tr>
<tr>
<td><code>path</code></td>
<td>Required. The path specification for the component whose drive name is to be returned.</td>
</tr>
</tbody>
</table>

**Remarks**

The `GetDriveName` method returns a zero-length string (""") if the drive can't be determined.

**Note** 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 following example illustrates the use of the `GetDriveName` method:

```javascript
function GetDriveLetter(path) {
  var fso, s = "";
  fso = new ActiveXObject("Scripting.FileSystemObject");
  s += fso.GetDrive(fso.GetDriveName(fso.GetAbsolutePathName(path)));
  return(s);
}
```
GetExtensionName Method

**Description**

Returns a string containing the extension name for the last component in a path.

**Syntax**

```javascript
object.GetExtensionName(path)
```

The `GetExtensionName` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <code>FileSystemObject</code>.</td>
</tr>
<tr>
<td><code>path</code></td>
<td>Required. The path specification for the component whose extension name is to be returned.</td>
</tr>
</tbody>
</table>

**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.

The following example illustrates the use of the `GetExtensionName` method:
function ShowExtensionName(filespec)
{
    var fso, s = "";
    fso = new ActiveXObject("Scripting.FileSystemObject");
    s += fso.GetExtensionName(filespec);
    return(s);
}
GetFile Method

**Description**

Returns a **File** object corresponding to the file in a specified path.

**Syntax**

```javascript
object.GetFile(filespec)
```

The `GetFile` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <strong>FileSystemObject</strong>.</td>
</tr>
<tr>
<td><code>filespec</code></td>
<td>Required. The <code>filespec</code> is the path (absolute or relative) to a specific file.</td>
</tr>
</tbody>
</table>

**Remarks**

An error occurs if the specified file does not exist.

The following example illustrates the use of the `GetFile` method:

```javascript
function ShowFileAccessInfo(filespec)
{
    var fso, f, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    f = fso.GetFile(filespec);
    s = f.OpenText("r");
    alert(s.read());
}
```

See Also  

Applies To  

Scripting Run-Time Reference  

Version 3
f = fso.GetFile(filespec);
s = f.Path.toUpperCase() + "<br>";
s += "Created: " + f.DateCreated + "<br>";
s += "Last Accessed: " + f.DateLastAccessed;
s += "Last Modified: " + f.DateLastModified
return(s);
}
GetFileName Method

See Also  Applies To

Description

Returns the last component of specified path that is not part of the drive specification.

Syntax

```
object.GetFileName(pathspec)
```

The `GetFileName` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <code>FileSystemObject</code>.</td>
</tr>
<tr>
<td><code>pathspec</code></td>
<td>Required. The path (absolute or relative) to a specific file.</td>
</tr>
</tbody>
</table>

Remarks

The `GetFileName` method returns a zero-length string (""") if `pathspec` does not end with the named component.

**Note** 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 following example illustrates the use of the `GetFileName` method:

```javascript
function ShowFile Name(filespec)
{
    var fso, s = "";
    fso = new ActiveXObject("Scripting.FileSystemObject");
    s += fso.GetFileName(filespec);
    return(s);
}
```
GetFileVersion Method

Description

Returns the version number of a specified file.

Syntax

```javascript
object.GetFileVersion(pathspec)
```

The `GetFileVersion` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <code>FileSystemObject</code>.</td>
</tr>
<tr>
<td><code>pathspec</code></td>
<td>Required. The path (absolute or relative) to a specific file.</td>
</tr>
</tbody>
</table>

Remarks

The `GetFileVersion` method returns a zero-length string (""") if `pathspec` does not end with the named file or if the file does not contain version information.

Note  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 following example illustrates the use of the `GetFileVersion` method:

```javascript
function ShowFileVersion(pathspec)
{
    var fso, s = "";
    fso = new ActiveXObject("Scripting.FileSystemObject");
    s += fso.GetFileVersion(pathspec);
    if (s == "")
        s = "No version information available.";
    return(s);
}
```
GetFolder Method

Description

Returns a **Folder** object corresponding to the folder in a specified path.

Syntax

```
object.GetFolder(folderspec)
```

The **GetFolder** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a <strong>FileSystemObject</strong>.</td>
</tr>
<tr>
<td>folderspec</td>
<td>Required. The <em>folderspec</em> is the path (absolute or relative) to a specific folder.</td>
</tr>
</tbody>
</table>

Remarks

An error occurs if the specified folder does not exist.

The following example illustrates the use of the **GetFolder** method:

```javascript
function ShowFolderList(folderspec) {
    var fso, f, fc, s;
```
fso = new ActiveXObject("Scripting.FileSystemObject");
f = fso.GetFolder(folderspec);
fcs = new Enumerator(f.SubFolders);
s = ""
for (; !fc.atEnd(); fc.moveNext())
{
    s += fc.item();
    s += "<br>"
}
return(s);
**GetParentFolderName Method**

**Description**

Returns a string containing the name of the parent folder of the last component in a specified path.

**Syntax**

```
object.GetParentFolderName(path)
```

The `GetParentFolderName` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <code>FileSystemObject</code>.</td>
</tr>
<tr>
<td><code>path</code></td>
<td>Required. The path specification for the component whose parent folder name is to be returned.</td>
</tr>
</tbody>
</table>

**Remarks**

The `GetParentFolderName` method returns a zero-length string (""") if there is no parent folder for the component specified in the `path` argument.

**Note** 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 following example illustrates the use of the `GetParentFolderName` method:

```javascript
function ShowParentFolderName(filespec)
{
    var fso, s = "";
    fso = new ActiveXObject("Scripting.FileSystemObject");
    s += fso.GetParentFolderName(filespec);
    return(s);
}
```
Microsoft® JScript®

**GetSpecialFolder**

Method

See Also  
Applies To

---

**Description**

Returns the special folder object specified.

**Syntax**

`object.GetSpecialFolder(folderspec)`

The `GetSpecialFolder` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <strong>FileSystemObject</strong>.</td>
</tr>
<tr>
<td><code>folderspec</code></td>
<td>Required. The name of the special folder to be returned. Can be any of the constants shown in the Settings section.</td>
</tr>
</tbody>
</table>

**Settings**

The `folderspec` argument can have any of the following values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>WindowsFolder</strong></td>
<td>0</td>
<td>The Windows folder contains files installed by the Windows operating system.</td>
</tr>
<tr>
<td><strong>SystemFolder</strong></td>
<td>1</td>
<td>The System folder contains libraries, fonts, and device drivers.</td>
</tr>
</tbody>
</table>
The Temp folder is used to store temporary files. Its path is found in the TMP environment variable.

The following example illustrates the use of the `GetSpecialFolder` method:

```javascript
var fso, tempfile;

fso = new ActiveXObject("Scripting.FileSystemObject");

function CreateTempFile()
{
    var tfolder, tfile, tname, fname, TemporaryFolder = 2;
    tfolder = fso.GetSpecialFolder(TemporaryFolder);
    tname = fso.GetTempName();
    tfile = tfolder.CreateTextFile(tname);
    return(tfile);
}

tempfile = CreateTempFile();
tempfile.writeline("Hello World");
tempfile.close();
```
**GetTempName Method**

**Description**

Returns a randomly generated temporary file or folder name that is useful for performing operations that require a temporary file or folder.

**Syntax**

```
object.GetTempName ( );
```

The optional `object` is always the name of a `FileSystemObject`.

**Remarks**

The `GetTempName` method does not create a file. It provides only a temporary file name that can be used with `CreateTextFile` to create a file.

The following example illustrates the use of the `GetTempName` method:

```javascript
var fso, tempfile;
fso = new ActiveXObject("Scripting.FileSystemObject");
function CreateTempFile()
```


{
    var tfolder, tfile, tname, fname, TemporaryFolder;
    tfolder = fso.GetSpecialFolder(TemporaryFolder);
    tname = fso.GetTempName();
    tfile = tfolder.CreateTextFile(tname);
    return(tfile);
}

tempfile = CreateTempFile();
tempfile.writeline("Hello World");
tempfile.close();
See Also  Applies To

Description

Returns **True** if the specified drive is ready; **False** if it is not.

Syntax

```
object.IsReady
```

The `object` is always a **Drive** object.

Remarks

For removable-media drives and CD-ROM drives, **IsReady** returns **True** only when the appropriate media is inserted and ready for access.

The following code illustrates the use of the **IsReady** property:

```
function ShowDriveInfo(drvpath) {
    var fso, d, s, t;
    fso = new ActiveXObject("Scripting.FileSystemObject")
    d = fso.GetDrive(drvpath)
    switch (d.DriveType)
```
{ 
    case 0: t = "Unknown"; break;
    case 1: t = "Removable"; break;
    case 2: t = "Fixed"; break;
    case 3: t = "Network"; break;
    case 4: t = "CD-ROM"; break;
    case 5: t = "RAM Disk"; break;
}

s = "Drive " + d.DriveLetter + ": " + t;

if (d.IsReady)
    s += "<br>" + "Drive is Ready.";
else
    s += "<br>" + "Drive is not Ready.";

return(s);
}
Microsoft® JScript® IsRootFolder Property

See Also  Applies To

Description

Returns True if the specified folder is the root folder; False if it is not.

Syntax

object.IsRootFolder

The object is always a Folder object.

Remarks

The following code illustrates the use of the IsRootFolder property:

```javascript
function DisplayLevelDepth(pathspec)
{
    var fso, f, n, s = "";
    fso = new ActiveXObject("Scripting.FileSystemObject");
    f = fso.GetFolder(pathspec);
    n = 0;
    if (f.IsRootFolder)
        s = "The specified folder is the root folder."
```
else
{
    do
    {
        f = f.ParentFolder;
        n++;
    }
    while (!f.IsRootFolder)
    
s = "The specified folder is nested " + n + " levels deep."
}
return(s);
}
**Description**

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

**Syntax**

```
object.Items()
```

The object is always the name of a Dictionary object.

**Remarks**

The following code illustrates use of the Items method:

```javascript
function ItemsDemo()
{
    var a, d, i, s;          // Create some variables.
    d = new ActiveXObject("Scripting.Dictionary");
    d.Add("a", "Athens");    // Add some keys.
    d.Add("b", "Belgrade");
    d.Add("c", "Cairo");
    a = (new VBArray(d.Items())).toArray();  // Get the array of items.
    s = "";
    for (i in a)               // Iterate the dictionary.
        s = s + a[i] + " :: " + d.Item(a[i]);
    s = s.substring(0, s.length - 4) + " = ";
    return s;
}
```

{
    s += a[i] + "<br>";
}
return(s); // Return the result.
**Keys Method**

**See Also**

**Applies To**

**Description**

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

**Syntax**

```
object.Keys()
```

The `object` is always the name of a **Dictionary** object.

**Remarks**

The following code illustrates use of the **Keys** method:

```javascript
function KeysDemo()
{
    var a, d, i, s; // Create some variables.
    d = new ActiveXObject("Scripting.Dictionary");
    d.Add("a", "Athens"); // Add some keys and items.
    d.Add("b", "Belgrade");
    d.Add("c", "Cairo");
    a = (new VBArray(d.Keys())).toArray(); // Get the keys.
    s = "";
```
for (i in a) // Iterate the dictionary
{
    s += a[i] + " - " + d(a[i]) + "<br>";
}
return(s); // Return the results
Line Property

Description

Read-only property that returns the current line number in a TextStream file.

Syntax

```
object.Line
```

The object is always the name of a TextStream object.

Remarks

After a file is initially opened and before anything is written, Line is equal to 1.

The following example illustrates the use of the Line property:

```
function GetLine()
{
    var fso, f, r
    var ForReading = 1, ForWriting =
    fso = new ActiveXObject("Scripti
f = fso.OpenTextFile("c:\textfile.txt", ForWriting, true)
f.WriteLine("Hello world!");
f.WriteLine("JScript is fun");
f.Close();
f = fso.OpenTextFile("c:\textfile.txt", ForReading);
r = f.ReadAll();
return(f.Line);
}
Move Method

See Also  Applies To

Description

Moves a specified file or folder from one location to another.

Syntax

\[ \text{object}.\text{Move}( \text{destination} ); \]

The **Move** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>object</strong></td>
<td>Required. Always the name of a <strong>File</strong> or <strong>Folder</strong> object.</td>
</tr>
<tr>
<td><strong>destination</strong></td>
<td>Required. Destination where the file or folder is to be moved. Wildcard characters are not allowed.</td>
</tr>
</tbody>
</table>

Remarks

The results of the **Move** method on a **File** or **Folder** are identical to operations performed using **FileSystemObject.MoveFile** or **FileSystemObject.MoveFolder**. You should note, however, that the alternative methods are capable of moving multiple files or folders.
**MoveFile Method**

**Description**

Moves one or more files from one location to another.

**Syntax**

```
object.MoveFile ( source, destination );
```

The **MoveFile** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a FileSystemObject.</td>
</tr>
<tr>
<td><code>source</code></td>
<td>Required. The path to the file or files to be moved.</td>
</tr>
<tr>
<td></td>
<td>The <code>source</code> argument string can contain wildcard characters in the last path component only.</td>
</tr>
<tr>
<td><code>destination</code></td>
<td>Required. The path where the file or files are to be moved.</td>
</tr>
<tr>
<td></td>
<td>The <code>destination</code> argument can't contain wildcard characters.</td>
</tr>
</tbody>
</table>

**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.

**Important** This method allows moving files between volumes only if supported by the operating system.

The following example illustrates the use of the **MoveFile** method:

```javascript
function MoveFile2Desktop(filespec) {
    var fso;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    fso.MoveFile(filespec, "c:\windows\desktop\")
}
```
Microsoft® JScript® MoveFolder Method

**See Also**

**Applies To**

---

**Description**

Moves one or more folders from one location to another.

**Syntax**

```
object.MoveFolder ( source, destination );
```

The `MoveFolder` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <strong>FileSystemObject</strong>.</td>
</tr>
<tr>
<td><code>source</code></td>
<td>Required. The path to the folder or folders to be moved. The <code>source</code> argument string can contain wildcard characters in the last path component only.</td>
</tr>
<tr>
<td><code>destination</code></td>
<td>Required. The path where the folder or folders are to be moved. The <code>destination</code> argument can't contain wildcard characters.</td>
</tr>
</tbody>
</table>

**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 folder to create. In either case, three things can happen when an individual
folder is moved:

- If \textit{destination} does not exist, the folder gets moved. This is the usual case.
- If \textit{destination} is an existing file, an error occurs.
- If \textit{destination} is a directory, an error occurs.

An error also occurs if a wildcard character that is used in \textit{source} doesn't match any folders. The \textbf{MoveFolder} method stops on the first error it encounters. No attempt is made to roll back any changes made before the error occurs.

\begin{footnotesize}
\begin{itemize}
    \item \textbf{Important} This method allows moving folders between volumes only if supported by the operating system.
\end{itemize}
\end{footnotesize}

The following example illustrates the use of the \textbf{MoveFolder} method:

\begin{verbatim}
function MoveFldr2Desktop(fldrspec)
{
    var fso;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    fso.MoveFolder(fldrspec, "c:\\windows\\desktop")
}
\end{verbatim}
Name Property

Description

Sets or returns the name of a specified file or folder. Read/write.

Syntax

_object_.Name [= newname]

The Name property has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a File or Folder object.</td>
</tr>
<tr>
<td>newname</td>
<td>Optional. If provided, newname is the new name of the specified object.</td>
</tr>
</tbody>
</table>

Remarks

The following code illustrates the use of the Name property:

```javascript
function ShowFileAccessInfo(filespec)
{
    var fso, f, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    f = fso.GetFile(filespec);
```
s = f.Name + " on Drive " + f.Drive + "<br>";
s += "Created: " + f.DateCreated + "<br>";
s += "Last Accessed: " + f.DateLastAccessed + "<br>";
s += "Last Modified: " + f.DateLastModified;
return(s);
}
Microsoft® JScript®

OpenTextFile

Method

See Also

Applies To

Description

Opens a specified file and returns a TextStream object that can be used to read from, write to, or append to the file.

Syntax

\texttt{object.OpenTextFile(filename[, iomode[, create[, format]]])}

The \texttt{OpenTextFile} method has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. \textit{Object} is always the name of a FileSystemObject.</td>
</tr>
<tr>
<td>filename</td>
<td>Required. \texttt{String expression} that identifies the file to open.</td>
</tr>
<tr>
<td>iomode</td>
<td>Optional. Can be one of three constants: \texttt{ForReading}, \texttt{ForWriting}, or \texttt{ForAppending}.</td>
</tr>
<tr>
<td>create</td>
<td>Optional. Boolean value that indicates whether a new file can be created if the specified \texttt{filename} doesn't exist. The value is \texttt{True} if a new file is created, \texttt{False} if it isn't created. If omitted, a new file isn't created.</td>
</tr>
<tr>
<td>format</td>
<td>Optional. One of three \texttt{Tristate} values used to indicate the format of the opened file. If omitted, the file is opened as ASCII.</td>
</tr>
</tbody>
</table>
Settings

The *iomode* argument can have any of the following settings:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForReading</td>
<td>1</td>
<td>Open a file for reading only. You can't write to this file.</td>
</tr>
<tr>
<td>ForWriting</td>
<td>2</td>
<td>Open a file for writing.</td>
</tr>
<tr>
<td>ForAppending</td>
<td>8</td>
<td>Open a file and write to the end of the file.</td>
</tr>
</tbody>
</table>

The *format* argument can have any of the following settings:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TristateTrue</td>
<td>Open the file as Unicode.</td>
</tr>
<tr>
<td>TristateFalse</td>
<td>Open the file as ASCII.</td>
</tr>
<tr>
<td>TristateUseDefault</td>
<td>Open the file using the system default.</td>
</tr>
</tbody>
</table>

Remarks

The following code illustrates the use of the *OpenTextFile* method to open a file for appending text:

```javascript
var fs, a, ForAppending;
ForAppending = 8;
fs = new ActiveXObject("Scripting.FileSystemObject");
a = fs.OpenTextFile("c:\testfile.txt", ForAppending, false);
...
a.Close();
```
**ParentFolder**

**Property**

<table>
<thead>
<tr>
<th>See Also</th>
<th>Applies To</th>
</tr>
</thead>
</table>

**Description**

Returns the folder object for the parent of the specified file or folder. Read-only.

**Syntax**

```javascript
object.ParentFolder
```

The `object` is always a `File` or `Folder` object.

**Remarks**

The following code illustrates the use of the `ParentFolder` property with a file:

```javascript
function ShowFileAccessInfo(filespec)
{
    var fso, f, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
f = fso.GetFile(filespec);
s = f.Name + " in " + f.ParentFolder + "<br>";
s += "Created: " + f.DateCreated + "<br>";
s += "Last Accessed: " + f.DateLastAccessed + "<b"
s += "Last Modified: " + f.DateLastModified;
return(s);
}
**Microsoft® JScript®**

**Path Property**

**Description**

Returns the path for a specified file, folder, or drive.

**Syntax**

```javascript
object.Path
```

The `object` is always a **File**, **Folder**, or **Drive** object.

**Remarks**

For drive letters, the root drive is not included. For example, the path for the C drive is C:, not C:\.

The following code illustrates the use of the **Path** property with a **File** object:

```javascript
function ShowFileAccessInfo(filespec)
{
    var fso, d, f, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    f = fso.GetFile(filespec);
    s = f.Path.toUpperCase() + "<br>";
```
s += "Created: " + f.DateCreated + "<br>
" + f.DateLastAccessed + "<br>
" + f.DateLastModified + "<br>
return(s);
Read Method

See Also

Applies To

Description

Reads a specified number of characters from a TextStream file and returns the resulting string.

Syntax

object.Read(characters)

The Read method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a TextStream object.</td>
</tr>
<tr>
<td>characters</td>
<td>Required. Number of characters you want to read from the file.</td>
</tr>
</tbody>
</table>

The following example illustrates how to use the Read method to read a six character header from a file and return the resulting string:

```javascript
function GetHeader()
{
    var fso, f;
```
var ForReading = 1, ForWriting = 2;

fso = new ActiveXObject("Scripting.FileSystemObject");
f = fso.OpenTextFile("c:\testfile.txt", ForWriting, true);
f.Write("Header");
f.Write("1234567890987654321");
f.Close();
f = fso.OpenTextFile("c:\testfile.txt");
return(f.Read(6));
**MethodImpl**

**See Also**

**Applies To**

---

**Description**

Reads an entire line (up to, but not including, the newline character) from a **TextStream** file and returns the resulting string.

**Syntax**

```
object.ReadLine( )
```

The `object` argument is always the name of a **TextStream** object.

**Remarks**

The following example illustrates the use of the **Line** property:

```javascript
function GetLine()
{
    var fso, f, r;
    var ForReading = 1, ForWriting = 2;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    f = fso.OpenTextFile("c:\testfile.txt", ForReading, true);
```
f.WriteLine("Hello world!");
f.WriteLine("JScript is fun");
f.Close();
f = fso.OpenTextFile("c:\\testfile.txt");
r = f.ReadLine();
return(r);
**Description**

Removes a key, item pair from a **Dictionary** object.

**Syntax**

\[ \text{object}.\text{Remove}(key) \]

The `Remove` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <strong>Dictionary</strong> object.</td>
</tr>
<tr>
<td><code>key</code></td>
<td>Required. Key associated with the key, item pair you want to remove from the <strong>Dictionary</strong> object.</td>
</tr>
</tbody>
</table>

**Remarks**

An error occurs if the specified key, item pair does not exist.

The following code illustrates use of the `Remove` method:

```javascript
var a, d, i, s; // Create some

// Add some
```
d.Add("b", "Belgrade");
d.Add("c", "Cairo");

... 

d.Remove("b"); // Remove
Returns a **Folder** object representing the root folder of a specified drive. Read-only.

**Syntax**

```javascript
object.RootFolder
```

The *object* is always a **Drive** object.

**Remarks**

All the files and folders contained on the drive can be accessed using the returned **Folder** object.

The following example illustrates the use of the **RootFolder** property:

```javascript
function GetRootFolder(drv)
{
  var fso,d;
  fso = new ActiveXObject("Scripting.FileSystemObject");
  if (fso.DriveExists(drv))
  {
```

d = fso.GetDrive(drv);
return(d.RootFolder);
}
else
    return(false);
}
**SerialNumber Property**

**Description**

Returns the decimal serial number used to uniquely identify a disk volume.

**Syntax**

```javascript
object.SerialNumber
```

The `object` is always a `Drive` object.

**Remarks**

You can use the `SerialNumber` property to ensure that the correct disk is inserted in a drive with removable media.

The following code illustrates the use of the `SerialNumber` property:

```javascript
function ShowDriveInfo(drvpath)
{
    var fso, d, s, t;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    d = fso.GetDrive(fso.GetDriveName(fso.GetAbsolutePathName(drvpath)));
    switch (d.DriveType)
```
```cpp
{
    case 0: t = "Unknown"; break;
    case 1: t = "Removable"; break;
    case 2: t = "Fixed"; break;
    case 3: t = "Network"; break;
    case 4: t = "CD-ROM"; break;
    case 5: t = "RAM Disk"; break;
}

s = "Drive " + d.DriveLetter + ": - " + t;
s += "<br>" + "SN: " + d.SerialNumber;
return(s);
```
ShareName Property

See Also  Applies To

Description

Returns the network share name for a specified drive.

Syntax

object.ShareName

The object is always a Drive object.

Remarks

If object is not a network drive, the ShareName property returns a zero-length string (""").

The following code illustrates the use of the ShareName property:

```javascript
function ShowDriveInfo(drvpath)
{
    var fso, d, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    d = fso.GetDrive(fso.GetDriveName(fso.GetAbsolutePathName(drvpath)));
    s = "Drive " + d.DriveLetter + ": - " + d.ShareName
    alert(s);
}
```
return(s);
}

**Microsoft® JScript® ShortName Property**

**See Also**

**Applies To**

---

**Description**

Returns the short name used by programs that require the earlier 8.3 naming convention.

**Syntax**

```javascript
object.ShortName
```

The `object` is always a `File` or `Folder` object.

**Remarks**

The following code illustrates the use of the `ShortName` property with a `File` object:

```javascript
function ShowShortName(filespec)
{
    var fso, f, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    f = fso.GetFile(filespec);
    s = "The short name for " + "" + f.Name;
    s += "" + "<br>";
    s += "is: " + "" + f.ShortName + "";
```

---
return(s);
}
Microsoft® JScript®

**ShortPath**

Property

See Also  
Applies To

---

**Description**

Returns the short path used by programs that require the earlier 8.3 file naming convention.

**Syntax**

```javascript
object.ShortPath
```

The `object` is always a **File** or **Folder** object.

**Remarks**

The following code illustrates the use of the **ShortName** property with a **File** object:

```javascript
function ShowShortPath(filespec)
{
    var fso, f, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    f = fso.GetFile(filespec);
    s = "The short path for " + "+" + f.Name;
    s += "+" + <br>
    s += "is: " + "" + f.ShortPath + ";

    // Display the result
    document.write(s);
}
```

return(s);
}


Description

For files, returns the size, in bytes, of the specified file. For folders, returns the size, in bytes, of all files and subfolders contained in the folder.

Syntax

```javascript
object.Size
```

The `object` is always a `File` or `Folder` object.

Remarks

The following code illustrates the use of the `Size` property with a `Folder` object:

```javascript
function ShowFolderSize(filespec)
{
    var fso, f, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    f = fso.GetFolder(filespec);
    s = f.Name + " uses " + f.size + " bytes.";
    return(s);
}
```
Skip Method

See Also

Applies To

Description

Skips a specified number of characters when reading a TextStream file.

Syntax

object.Skip(characters)

The Skip method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a TextStream object.</td>
</tr>
<tr>
<td>characters</td>
<td>Required. Number of characters to skip when reading a file.</td>
</tr>
</tbody>
</table>

Remarks

Skipped characters are discarded.

The following example illustrates the use of the Skip method:

```javascript
function SkipDemo()
{
```
var fso, f, r;
var ForReading = 1, ForWriting = 2;

fso = new ActiveXObject("Scripting.FileSystemObject")
f = fso.OpenTextFile("c:\\testfile.txt", ForWriting, true);
f.WriteLine("Hello world!");
f.WriteLine("JScript is fun");
f.Close();

f = fso.OpenTextFile("c:\\testfile.txt");
f.Skip(6);
r = f.ReadLine();
return(r);
}

________________________________________
SubFolders Property

See Also

Applies To

Description

Returns a **Folders** collection consisting of all folders contained in a specified folder, including those with hidden and system file attributes set.

Syntax

```
object.SubFolders
```

The `object` is always a **Folder** object.

Remarks

The following code illustrates the use of the **SubFolders** property:

```
function ShowFolderList(folderspec)
{
    var fso, f, fc, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
f = fso.GetFolder(folderspec);
fcs = new Enumerator(f.SubFolders);
s = "";
```
for (;!fc.atEnd(); fc.moveNext())
{
    s += fc.item();
    s += "<br>";
}

return(s);
**TextStream Object**

**Description**

Facilitates sequential access to file.

**Syntax**

```
TextStream.{property | method( )}
```

The *property* and *method* arguments can be any of the properties and methods associated with the TextStream object. Note that in actual usage, TextStream is replaced by a variable placeholder representing the TextStream object returned from the FileSystemObject.

**Remarks**

In the following code, `a` is the TextStream object returned by the CreateTextFile method on the FileSystemObject:

```javascript
var fso = new ActiveXObject("Scripting.FileSystemObject");
var a = fso.CreateTextFile("c:\testfile.txt", true);
a.WriteLine("This is a test.");
a.Close();
```

**WriteLine** and **Close** are two methods of the TextStream object.
Microsoft® JScript® **TotalSize** Property

**See Also**   **Applies To**

---

**Description**

Returns the total space, in bytes, of a drive or network share.

**Syntax**

```
object.TotalSize
```

The `object` is always a `Drive` object.

**Remarks**

The following code illustrates the use of the `TotalSize` property:

```javascript
function SpaceReport(drvPath)
{
    var fso, d, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    d = fso.GetDrive(fso.GetDriveName(drvPath));
    s = "Drive " + drvPath + " - ";
    s += d.VolumeName + "<br>";
    s += "Total Space: " + d.TotalSize/1024 + " Kbytes<br>
    s += "Free Space: " + d.FreeSpace/1024 + " Kbytes"
}
```
return(s);
}

### Type Property

**Description**

Returns information about the type of a file or folder. For example, for files ending in .TXT, "Text Document" is returned.

**Syntax**

```javascript
object.Type
```

The `object` is always a File or Folder object.

**Remarks**

The following code illustrates the use of the Type property to return a folder type. In this example, try providing the path of the Recycle Bin or other unique folder to the procedure.

```javascript
function ShowFileType(filespec) {
    var fso, f, s;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    if (fso.FolderExists(filespec))
        f = fso.GetFolder(filespec);
    else if (fso.FileExists(filespec))
```
f = fso.GetFile(filespec);
else
    s = "File or Folder does not exist."
else
    s = f.Name + " is a " + f.Type;
return(s);
}
**VolumeName**

**Property**

Sets or returns the volume name of the specified drive. Read/write.

### Syntax

```
object.VolumeName [= newname]
```

The **VolumeName** property has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a Drive object.</td>
</tr>
<tr>
<td><code>newname</code></td>
<td>Optional. If provided, <code>newname</code> is the new name of the specified <code>object</code>.</td>
</tr>
</tbody>
</table>

### Remarks

The following code illustrates the use of the **VolumeName** property:

```javascript
function SpaceReport(drvPath)
{
    var fso, d, s;
    fso = new ActiveXObject("Scripting.FileSystemObject"),
    d = fso.GetDrive(drvPath); 
    s = d.VolumeName; 
    return s;
}
```


d = fso.GetDrive(fso.GetDriveName(drvPath));
s = "Drive " + drvPath + " - ";
s += d.VolumeName + "<br>");
s += "Total Space: " + d.TotalSize/1024 + " Kbytes <l
s += "Free Space: " + d.FreeSpace/1024 + " Kbytes"
return(s);
}
**Write Method**

**Description**

Writes a specified string to a **TextStream** file.

**Syntax**

```
object.Write(string)
```

The **Write** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a <strong>TextStream</strong> object.</td>
</tr>
<tr>
<td>string</td>
<td>Required. The text you want to write to the file.</td>
</tr>
</tbody>
</table>

**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.

The following example illustrates the use of the **Write** method:

```javascript
function WriteDemo()
```
```javascript
{
    var fso, f, r
    var ForReading = 1, ForWriting = 2;
    fso = new ActiveXObject("Scripting.FileSystemObject")
    f = fso.OpenTextFile("c:\\testfile.txt", ForWriting, true)
    f.Write("Hello world!");
    f.Close();
    f = fso.OpenTextFile("c:\\testfile.txt", ForReading);
    r = f.ReadLine();
    return(r);
}
```
WriteBlankLines Method

Description

Writes a specified number of newline characters to a TextStream file.

Syntax

object.WriteBlankLines(lines)

The WriteBlankLines method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a TextStream object.</td>
</tr>
<tr>
<td>lines</td>
<td>Required. Number of newline characters you want to write to the file.</td>
</tr>
</tbody>
</table>

Remarks

The following example illustrates the use of the WriteBlankLines method:

```javascript
function WriteBlanksDemo()
{
```
var fso, f, r;
var ForReading = 1, ForWriting = 2;
fso = new ActiveXObject("Scripting.FileSystemObject");
f = fso.OpenTextFile("c:\testfile.txt", ForWriting, true);
f.Write("Hello world!");
f.WriteBlankLines(2);
f.Write("JScript is fun!");
f.Close();
f = fso.OpenTextFile("c:\testfile.txt");
r = f.ReadAll();
return(r);
**WriteLine** Method

**Description**

Writes a specified string and newline character to a `TextStream` file.

**Syntax**

```
object.WriteLine([string])
```

The `WriteLine` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <code>TextStream</code> object.</td>
</tr>
<tr>
<td><code>string</code></td>
<td>Optional. The text you want to write to the file. If omitted, a newline character is written to the file.</td>
</tr>
</tbody>
</table>

**Remarks**

The following example illustrates use of the `WriteLine` method:

```javascript
var fso, f;
   fso = new ActiveXObject("Scriptin
f = fso.CreateTextFile("c:\\testfile.txt");
f.WriteLine("This is a test.");
f.Close();
Copyright

Microsoft® JScript®

Information in this document is subject to change without notice. The names of companies, products, people, characters, and/or data mentioned herein are fictitious and are in no way intended to represent any real individual, company, product, or event, unless otherwise noted. Complying with all applicable copyright laws is the responsibility of the user. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 1991-2000 Microsoft Corporation. All rights reserved.

Microsoft, MS, MS-DOS, ActiveX, JScript, Microsoft Press, Visual Basic, Windows, Windows NT, Win32, and Win32s are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries.

Other product and company names mentioned herein may be the trademarks of their respective owners.
**Description**

Returns an initial value of `Number.POSITIVE_INFINITY`.

**Syntax**

```
Infinity
```

**Remarks**

The `Infinity` property is a member of the `Global` object, and is made available when the scripting engine is initialized.
**getDescription**

**See Also**  
**Applies To**

**Description**

Returns the day of the month value in a **Date** object using **local time**.

**Syntax**

```javascript
objDate.getDate()
```

**Remarks**

To get the date value using **Universal Coordinated Time (UTC)**, use the **getUTCDate** method.

The return value is an integer between 1 and 31 that represents the date value in the **Date** object.

The following example illustrates the use of the **getDate** method:

```javascript
function DateDemo()
{
    var d, s = "Today's date is: ";
    d = new Date();
    s += (d.getMonth() + 1) + "/";
    s += d.getDate() + "/";
}
s += d.getYear();
return(s);
}
**getDay Method**

**See Also**

**Applies To**

---

**Description**

Returns the day of the week value in a **Date** object using local time.

**Syntax**

```javascript
objDate.getDay()
```

**Remarks**

To get the day using **Universal Coordinated Time (UTC)**, use the **getUTCDay** method.

The value returned from the **getDay** method is an integer between 0 and 6 representing the day of the week and corresponds to a day of the week as follows:

- 0 = Sunday
- 1 = Monday
- 2 = Tuesday
- 3 = Wednesday
- 4 = Thursday
- 5 = Friday
- 6 = Saturday

The following example illustrates the use of the **getDay** method:

```javascript
function DateDemo()
```
function day() {
    var d, day, x, s = "Today is: ";
    var x = new Array("Sunday", "Monday", "Tuesday");
    var x = x.concat("Wednesday", "Thursday", "Friday");
    var x = x.concat("Saturday");
    d = new Date();
    day = d.getDay();
    return(s += x[day]);
}


**getHours Method**

**See Also**

**Applies To**

---

**Description**

Returns the hours value in a **Date** object using **local time**.

**Syntax**

`objDate.getHours()`

**Remarks**

To get the hours value using **Universal Coordinated Time (UTC)**, use the **getUTCHours** method.

The **getHours** method returns an integer between 0 and 23 indicating the number of hours since midnight. A zero occurs in two situations: the time is before 1:00:00 am, or the time was not stored in the **Date** object when the object was created. The only way to determine which situation you have is to also check the minutes and seconds for zero values. If they are all zeroes, it is nearly certain that the time was not stored in the **Date** object.

The following example illustrates the use of the **getHours** method:

```javascript
function TimeDemo()
{
    var d, s = "The current local time is: ";
    var c = ":";
```
d = new Date();
s += d.getHours() + c;
s += d.getMinutes() + c;
s += d.getSeconds() + c;
s += d.getMilliseconds();
return(s);
}
Microsoft® JScript®

getMinutes

Method

See Also

Applies To

Description

Returns the minutes value in a **Date** object using local time.

Syntax

```javascript
objDate.getMinutes()
```

Remarks

To get the minutes value using **Universal Coordinated Time (UTC)**, use the **getUTCMinutes** method.

The `getMinutes` method returns an integer between 0 and 59 equal to the minutes value stored in the **Date** object. A zero is returned in two situations: one occurs when the time is less than one minute after the hour. The other occurs when the time was not stored in the **Date** object when the object was created. The only way to determine which situation you have is to also check the hours and seconds for zero values. If they are all zeroes, it is nearly certain that the time was not stored in the **Date** object.

The following example illustrates the use of the `getMinutes` method:

```javascript
function TimeDemo()
{
    var d, s = "The current local time is:",&
    var c = ":";
```
d = new Date();
s += d.getHours() + c;
s += d.getMinutes() + c;
s += d.getSeconds() + c;
s += d.getMilliseconds();
return(s);
}

Microsoft® JScript®

**getMonth Method**

**See Also**

**Applies To**

---

**Description**

Returns the month value in the **Date** object using local time.

**Syntax**

`objDate.getMonth()`

**Remarks**

To get the month value using **Universal Coordinated Time (UTC)**, use the **getUTCMonth** method.

The **getMonth** method returns an integer between 0 and 11 indicating the month value in the **Date** object. The integer returned is not the traditional number used to indicate the month. It is one less. If "Jan 5, 1996 08:47:00" is stored in a **Date** object, **getMonth** returns 0.

The following example illustrates the use of the **getMonth** method:

```javascript
function DateDemo()
{
    var d, s = "Today's date is: ";
    d = new Date();
    s += (d.getMonth() + 1) + "/";
}```
s += d.getDate() + "/";

s += d.getYear();

return(s);
}

**getSeconds Method**

**See Also**  
**Applies To**

---

**Description**

Returns the seconds value in a **Date** object using **local time**.

**Syntax**

```javascript
objDate.getSeconds()
```

**Remarks**

To get the seconds value using **Universal Coordinated Time (UTC)**, use the **getUTCSeconds** method.

The **getSeconds** method returns an integer between 0 and 59 indicating the seconds value of the indicated **Date** object. A zero is returned in two situations. One occurs when the time is less than one second into the current minute. The other occurs when the time was not stored in the **Date** object when the object was created. The only way to determine which situation you have is to also check the hours and minutes for zero values. If they are all zeroes, it is nearly certain that the time was not stored in the **Date** object.

The following example illustrates the use of the **getSeconds** method:

```javascript
function TimeDemo()
{
    var d, s = "The current local time:
";
```
var c = ":";
d = new Date();
s += d.getHours() + c;
s += d.getMinutes() + c;
s += d.getSeconds() + c;
s += d.getMilliseconds();
return(s);
}
Description

Returns the time value in a Date object.

Syntax

objDate.getTime()

Remarks

The getTime method returns an integer value representing the number of milliseconds between midnight, January 1, 1970 and the time value in the Date object. The range of dates is approximately 285,616 years from either side of midnight, January 1, 1970. Negative numbers indicate dates prior to 1970.

When doing multiple date and time calculations, it is frequently useful to define variables equal to the number of milliseconds in a day, hour, or minute. For example:

```javascript
var MinMilli = 1000 * 60
var HrMilli = MinMilli * 60
var DyMilli = HrMilli * 24
```

The following example illustrates the use of the getTime method.
function getTimeTest() {
    var d, s, t;
    var MinMilli = 1000 * 60;
    var HrMilli = MinMilli * 60;
    var DyMilli = HrMilli * 24;
    d = new Date();
    t = d.getTime();
    s = "It's been "
    s += Math.round(t / DyMilli) + " days since 1/1/70";
    return(s);
}
getTimezoneOffset Method

Description

Returns the difference in minutes between the time on the host computer and Universal Coordinated Time (UTC).

Syntax

`objDate.getTimezoneOffset()`

Remarks

The `getTimezoneOffset` method returns an integer value representing the number of minutes between the time on the current machine and UTC. These values are appropriate to the computer the script is executed on. If it is called from a server script, the return value is appropriate to the server. If it is called from a client script, the return value is appropriate to the client.

This number will be positive if you are behind UTC (e.g., Pacific Daylight Time), and negative if you are ahead of UTC (e.g., Japan).

For example, suppose a server in New York City is contacted by a client in Los Angeles on December 1. `getTimezoneOffset` returns 480 if executed on the client, or 300 if executed on the server.

The following example illustrates the use of the `getTimezoneOffset` method:

```javascript
function TZDemo()
```
{
    var d, tz, s = "The current local time is ";
    d = new Date();
    tz = d.getTimezoneOffset();
    if (tz < 0)
        s += tz / 60 + " hours before GMT";
    else if (tz == 0)
        s += "GMT";
    else
        s += tz / 60 + " hours after GMT";
    return(s);
}

**getDescription**

Returns the year value in a **Date** object.

**Syntax**

```javascript
objDate.getYear()
```

**Remarks**

This method is obsolete, and is provided for backwards compatibility only. Use the **getFullYear** method instead.

For years from 1900 through 1999, the year is a 2-digit integer value returned as the difference between the stored year and 1900. For other dates, the 4-digit year is returned. For example, 1996 is returned as 96, but 1825 and 2025 are returned as-is.

**Note** For JScript version 1.0, **getYear** returns a value that is the result of the subtraction of 1900 from the year value in the provided **Date** object, regardless of the value of the year. For example, the year 1899 is returned as -1 and the year to 2000 is returned as 100.

The following example illustrates the use of the **getYear** method:

```javascript
function DateDemo()
{
}```
```javascript
var d, s = "Today's date is: ";
d = new Date();
s += (d.getMonth() + 1) + "/";
s += d.getDate() + "/";
s += d.getFullYear();
return(s);
}
```
Microsoft® JScript® getUTCFullYear Method

Description

Returns the year value in a Date object using Universal Coordinated Time (UTC).

Syntax

objDate.getFullYear()

Remarks

To get the year using local time, use the getFullYear method.

The getUTCFullYear method returns the year as an absolute number. This avoids the classic year 2000 problem where dates beginning with January 1, 2000 are confused with those beginning January 1, 1900.

The following example illustrates the use of the getUTCFullYear method:

```javascript
function UTCDateDemo()
{
    var d, s = "Today's UTC date is: ";
    d = new Date();
    s += (d.getUTCMonth() + 1) + "/";
```
s += d.getUTCDate() + "/";
    s += d.getUTCFullYear();
    return(s);
}
**getUTCHours Method**

**Description**

Returns the hours value in a Date object using Universal Coordinated Time (UTC).

**Syntax**

```
objDate.getUTCHours()
```

**Remarks**

To get the number of hours elapsed since midnight using local time, use the `getHours` method.

The `getUTCHours` method returns an integer between 0 and 23 indicating the number of hours since midnight. A zero occurs in two situations: the time is before 1:00:00 A.M., or a time was not stored in the `Date` object when the object was created. The only way to determine which situation you have is to also check the minutes and seconds for zero values. If they are all zeroes, it is nearly certain that the time was not stored in the `Date` object.

The following example illustrates the use of the `getUTCHours` method:

```javascript
function UTCTimeDemo()
{

```
var d, s = "Current Universal Coordinated Time (UTC) is:");
var c = ":";
d = new Date();
s += d.getUTCHours() + c;
s += d.getUTCMilliseconds();
s += d.getUTCMilliseconds();
return(s);"}
getUTCMinutes Method

Description

Returns the minutes value in a Date object using Universal Coordinated Time (UTC).

Syntax

objDate.getUTCMinutes()

Remarks

To get the number of minutes stored using local time, use the getMinutes method.

The getUTCMinutes method returns an integer between 0 and 59 equal to the number of minutes value in the Date object. A zero occurs in two situations: the time is less than one minute after the hour, or a time was not stored in the Date object when the object was created. The only way to determine which situation you have is to also check the hours and seconds for zero values. If they are all zeroes, it is nearly certain that the time was not stored in the Date object.

The following example illustrates the use of the getUTCMinutes method:

```javascript
function UTCTimeDemo()
{

```
var d, s = "Current Universal Coordinated Time (UTC) is:
var c = ":";
d = new Date();
s += d.getUTCHours() + c;
s += d.getUTCMinutes() + c;
s += d.getUTCSeconds() + c;
s += d.getUTCMilliseconds();
return(s);
}
Microsoft® JScript® **getUTC Month**

**Method**

**See Also**

**Applies To**

---

**Description**

Returns the month value value in a **Date** object using **Universal Coordinated Time (UTC)**.

**Syntax**

```
objDate.getUTC Month()
```

**Remarks**

To get the month in **local time**, use the **get Month** method.

The **getUTC Month** method returns an integer between 0 and 11 indicating the month value in the Date object. The integer returned is not the traditional number used to indicate the month. It is one less. If "Jan 5, 1996 08:47:00.0" is stored in a **Date** object, **getUTC Month** returns 0.

The following example illustrates the use of the **getUTC Month** method:

```javascript
function UTCDateDemo()
{
    var d, s = "Today's UTC date is: ";
    d = new Date();
```
s += (d.getUTCMonth() + 1) + "/";
s += d.getUTCDate() + "/";
s += d.getUTCFullYear();
return(s);
getUTCSeconds Method

Description

Returns the seconds value in a Date object using Universal Coordinated Time (UTC).

Syntax

objDate.getUTCSeconds()

Remarks

To get the number of seconds in local time, use the getSeconds method.

The getUTCSeconds method returns an integer between 0 and 59 indicating the seconds value of the indicated Date object. A zero occurs in two situations: the time is less than one second into the current minute, or a time was not stored in the Date object when the object was created. The only way to determine which situation you have is to also check the minutes and hours for zero values. If they are all zeroes, it is nearly certain that the time was not stored in the Date object.

The following example illustrates the use of the getUTCSeconds method:

```javascript
function UTCTimeDemo()
{
}```
var d, s = "Current Universal Coordinated Time (UTC) is:";
var c = ":";
d = new Date();
s += d.getUTCHours() + c;
s += d.getUTCMinutes() + c;
s += d.getUTCSeconds() + c;
s += d.getUTCMilliseconds();
return(s);
}
**setDate Method**

**Description**

Sets the numeric date of the **Date** object using local time.

**Syntax**

```javascript
objDate.setDate(numDate)
```

The `numDate` argument is a numeric value equal to the numeric date.

**Remarks**

To set the date value using **Universal Coordinated Time (UTC)**, use the `setUTCDate` method.

If the value of `numDate` is greater than the number of days in the month stored in the **Date** object or is a negative number, the date is set to a date equal to `numDate` minus the number of days in the stored month. For example, if the stored date is January 5, 1996, and `setDate(32)` is called, the date changes to February 1, 1996. Negative numbers have a similar behavior.

The following example illustrates the use of the `setDate` method:

```javascript
function SetDateDemo(newdate) {
    var d, s;
    d = new Date();
```
d.setDate(newdate);
s = "Current setting is ";
s += d.toLocaleString();
return(s);
}
**setHours Method**

**Description**

Sets the hour value in the **Date** object using local time.

**Syntax**

\[ \text{objDate.setHours}(\text{numHours}, [\text{numMin}, [\text{numSec}, \text{numMilli}]])) \]

The `setHours` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>numHours</code></td>
<td>Required. A numeric value equal to the hours value.</td>
</tr>
<tr>
<td><code>numMin</code></td>
<td>Optional. A numeric value equal to the minutes value. Must be supplied if either of the following arguments are used.</td>
</tr>
<tr>
<td><code>numSec</code></td>
<td>Optional. A numeric value equal to the seconds value. Must be supplied if the following argument is used.</td>
</tr>
<tr>
<td><code>numMilli</code></td>
<td>Optional. A numeric value equal to the milliseconds value.</td>
</tr>
</tbody>
</table>

**Remarks**

See Also

Applies To
All set methods taking optional arguments use the value returned from corresponding get methods, if you do not specify an optional argument. For example, if the numMonth argument is optional, but not specified, JScript uses the value returned from the getMonth method.

To set the hours value using Universal Coordinated Time (UTC), use the setUTCHours method.

If the value of an argument is greater than its range or is a negative number, other stored values are modified accordingly. For example, if the stored date is "Jan 5, 1996 00:00:00", and setHours(30) is called, the date is changed to "Jan 6, 1996 06:00:00." Negative numbers have a similar behavior.

The following example illustrates the use of the setHours method:

```
function SetHoursDemo(nhr, nmin, nsec) {
  var d, s;
  var sep = "":";
  d = new Date();
  d.setHours(nhr, nmin, nsec);
  s = "Current setting is " + d.toLocaleString();
  return(s);
}
```
**Description**

Sets the month value in the **Date** object using local time.

**Syntax**

```
objDate.setMonth(numMonth[, dateVal])
```

The `setMonth` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>numMonth</code></td>
<td>Required. A numeric value equal to the month.</td>
</tr>
<tr>
<td><code>dateVal</code></td>
<td>Optional. A numeric value representing the date. If not supplied, the value from a call to the <code>getDate</code> method is used.</td>
</tr>
</tbody>
</table>

**Remarks**

To set the month value using **Universal Coordinated Time (UTC)**, use the `setUTCMonth` method.

If the value of `numMonth` is greater than 11 (January is month 0) or is a negative number, the stored year is modified accordingly. For example, if the stored date is "Jan 5, 1996" and `setMonth(14)` is called, the date is changed to "Mar 5, 1997."

The following example illustrates the use of the `setMonth` method:
function SetMonthDemo(newmonth) {
    var d, s;
    d = new Date();
    d.setMonth(newmonth);
    s = "Current setting is ";
    s += d.toLocaleString();
    return(s);
}
Description

Sets the seconds value in the **Date** object using **local time**.

Syntax

```javascript
objDate.setSeconds(numSeconds[, numMilli])
```

The `setSeconds` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>numSeconds</code></td>
<td>Required. A numeric value equal to the seconds value.</td>
</tr>
<tr>
<td><code>numMilli</code></td>
<td>Optional. A numeric value equal to the milliseconds value.</td>
</tr>
</tbody>
</table>

Remarks

All **set** methods taking optional arguments use the value returned from corresponding **get** methods, if you do not specify an optional argument. For example, if the `numMonth` argument is optional, but not specified, JScript uses the value returned from the `getMonth` method.

To set the seconds value using **Universal Coordinated Time (UTC)**, use the **setUTCSeconds** method.

If the value of an argument is greater than its range or is a negative number, other stored
values are modified accordingly. For example, if the stored date is "Jan 5, 1996
00:00:00" and setSeconds(150) is called, the date is changed to "Jan 5, 1996 00:02:30."

The following example illustrates the use of the setSeconds method:

```javascript
function SetSecondsDemo(nsec, nmsec) {
    var d, s;
    var sep = "\:";
    d = new Date();
    d.setSeconds(nsec, nmsec);
    s = "Current setting is ";
    s += d.toLocaleString() + sep + d.getMilliseconds();
    return(s);
}
```
Description

Sets the date and time value in the `Date` object.

Syntax

```
objDate.setTime(milliseconds)
```

The `milliseconds` argument is an integer value representing the number of elapsed seconds since midnight, January 1, 1970 GMT.

Remarks

If `milliseconds` is negative, it indicates a date before 1970. The range of available dates is approximately 285,616 years from either side of 1970.

Setting the date and time with the `setTime` method is independent of the time zone.

The following example illustrates the use of the `setTime` method:

```javascript
function SetTimeTest(newtime)
{
    var d, s;
    d = new Date();
```
d.setTime(newtime);
s = "Current setting is ";
s += d.toUTCString();
return(s);
}
**Description**

Sets the year value in the **Date** object.

**Syntax**

```javascript
objDate.setYear(numYear)
```

The `numYear` argument is a numeric value equal to the year minus 1900.

**Remarks**

This method is obsolete, and is maintained for backwards compatibility only. Use the **setFullYear** method instead.

To set the year of a **Date** object to 1997, call **setYear(97)**. To set the year to 2010, call **setYear(2010)**. Finally, to set the year to a year in the range 0-99, use the **setFullYear** method.

**Note** For JScript version 1.0, **setYear** uses a value that is the result of the addition of 1900 to the year value provided by the `numYear`, regardless of the value of the year. For example, to set the year to 1899 `numYear` is -1 and to set the year to 2000 `numYear` is 100.

The following example illustrates the use of the **setYear** method:

```javascript
function SetYearDemo(newyear)
```
{ 
    var d, s;
    d = new Date();
    d.setYear(newyear);
    s = "Current setting is ";
    s += d.toLocaleString();
    return(s);
}

**setUTCFullYear Method**

**Description**

Sets the year value in the **Date** object using **Universal Coordinated Time (UTC)**.

**Syntax**

```
objDate.setUTCFullYear(numYear[, numMonth[, numDate]])
```

The **setUTCFullYear** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numYear</td>
<td>Required. A numeric value equal to the year.</td>
</tr>
<tr>
<td>numMonth</td>
<td>Optional. A numeric value equal to the month. Must be supplied if numDate is supplied.</td>
</tr>
<tr>
<td>numDate</td>
<td>Optional. A numeric value equal to the date.</td>
</tr>
</tbody>
</table>

**Remarks**

All **set** methods taking optional arguments use the value returned from corresponding **get** methods, if you do not specify an optional argument. For example, if the **numMonth** argument is optional, but not specified, JScript uses the value returned from the **getMonth** method.

In addition, if the value of an argument is greater that its range or is a
negative number, other stored values are modified accordingly.

To set the year using local time, use the `setFullYear` method.

The range of years supported in the `Date` object is approximately 285,616 years from either side of 1970.

The following example illustrates the use of the `setUTCFullYear` method:

```javascript
function SetUTCFullYearDemo(newyear) {
    var d, s;
    d = new Date();
    d.setUTCFullYear(newyear);
    s = "Current setting is ";
    s += d.toUTCString();
    return(s);
}
```
**setUTCHours**

**Method**

**Description**

Sets the hours value in the **Date** object using **Universal Coordinated Time (UTC)**.

**Syntax**

```javascript
objDate.setUTCHours(numHours[, numMin[, numSec[, numMilli]]])
```

The **setUTCHours** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>numHours</strong></td>
<td>Required. A numeric value equal to the hours value.</td>
</tr>
<tr>
<td><strong>numMin</strong></td>
<td>Optional. A numeric value equal to the minutes value. Must be supplied if either <strong>numSec</strong> or <strong>numMilli</strong> are used.</td>
</tr>
<tr>
<td><strong>numSec</strong></td>
<td>Optional. A numeric value equal to the seconds value. Must be supplied if <strong>numMilli</strong> argument is used.</td>
</tr>
<tr>
<td><strong>numMilli</strong></td>
<td>Optional. A numeric value equal to the milliseconds value.</td>
</tr>
</tbody>
</table>

**Remarks**

All **set** methods taking optional arguments use the value returned
from corresponding get methods, if you do not specify an optional argument. For example, if the numMonth argument is optional, but not specified, JScript uses the value returned from the getMonth method.

To set the hours value using local time, use the setHours method.

If the value of an argument is greater than its range or is a negative number, other stored values are modified accordingly. For example, if the stored date is "Jan 5, 1996 00:00:00.00", and setUTCHours(30) is called, the date is changed to "Jan 6, 1996 06:00:00.00."

The following example illustrates the use of the setUTCHours method:

```javascript
function SetUTCHoursDemo(nhr, nmin, nsec)
{
    var d, s;
    var sep = "":";
    d = new Date();
    d.setUTCHours(nhr, nmin, nsec);
    s = "Current setting is " + d.toUTCString()
    return(s);
}
```
**setUTCMinutes** Method

### See Also

### Applies To

### Description

Sets the minutes value in the **Date** object using **Universal Coordinated Time (UTC)**.

### Syntax

```javascript
objDate.setUTCMinutes([numMinutes[, numSeconds[, numMilli]]])
```

The **setUTCMinutes** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>numMinutes</strong></td>
<td>Required. A numeric value equal to the minutes value.</td>
</tr>
<tr>
<td><strong>numSeconds</strong></td>
<td>Optional. A numeric value equal to the seconds value. Must be supplied if <strong>numMilli</strong> is used.</td>
</tr>
<tr>
<td><strong>numMilli</strong></td>
<td>Optional. A numeric value equal to the milliseconds value.</td>
</tr>
</tbody>
</table>

### Remarks

All **set** methods taking optional arguments use the value returned from corresponding **get** methods, if you do not specify an optional argument. For example, if the **numMonth** argument is optional, but not specified, JScript uses the value returned from
the `getMonth` method.

To modify the minutes value using local time, use the `setMinutes` method.

If the value of an argument is greater than its range or is a negative number, other stored values are modified accordingly. For example, if the stored date is "Jan 5, 1996 00:00:00.00", and `setUTCMinutes(70)` is called, the date is changed to "Jan 5, 1996 01:10:00.00."

The following example illustrates the use of the `setUTCMinutes` method:

```javascript
function SetUTCMinutesDemo(nmin, nsec) {
  var d, s;
  var sep = "":";
  d = new Date();
  d.setUTCMinutes(nmin,nsec);
  s = "Current setting is " + d.toUTCString();
  return(s);
}
```
Microsoft® JScript® setUTC Month

Method

See Also  Applies To

Description

Sets the month value in the Date object using Universal Coordinated Time (UTC).

Syntax

`objDate.setUTC Month(numMonth[, dateVal])`

The setUTC Month method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numMonth</td>
<td>Required. A numeric value equal to the month.</td>
</tr>
<tr>
<td>dateVal</td>
<td>Optional. A numeric value representing the date. If not supplied, the value from a call to the getUTCDate method is used.</td>
</tr>
</tbody>
</table>

Remarks

To set the month value using local time, use the setMonth method.

If the value of numMonth is greater than 11 (January is month 0) or is a negative number, the stored year is incremented or decremented appropriately. For example, if the stored date is "Jan 5, 1996 00:00:00.00", and setUTC Month(14) is called, the date is changed to "Mar 5, 1997"
The following example illustrates the use of the `setUTCMonth` method:

```javascript
function SetUTCMonthDemo(newmonth) {
    var d, s;
    d = new Date();
    d.setUTCMonth(newmonth);
    s = "Current setting is ";
    s += d.toUTCString();
    return(s);
}
```
**setUTCSeconds** Method

**See Also**: Applies To

### Description

Sets the seconds value in the **Date** object using **Universal Coordinated Time (UTC)**.

### Syntax

```javascript
objDate.setUTCSeconds(numSeconds[, numMilli])
```

The **setUTCSeconds** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numSeconds</td>
<td>Required. A numeric value equal to the seconds value.</td>
</tr>
<tr>
<td>numMilli</td>
<td>Optional. A numeric value equal to the milliseconds value.</td>
</tr>
</tbody>
</table>

### Remarks

All **set** methods taking optional arguments use the value returned from corresponding **get** methods, if you do not specify an optional argument. For example, if the **numMonth** argument is optional, but not specified, JScript uses the value returned from the **getMonth** method.

To set the seconds value using **local time**, use the **setSeconds** method.
If the value of an argument is greater than its range or is a negative number, other stored values are modified accordingly. For example, if the stored date is "Jan 5, 1996 00:00:00.00" and `setSeconds(150)` is called, the date is changed to "Jan 5, 1996 00:02:30.00."

The following example illustrates the use of the `setSeconds` method:

```javascript
function SetUTCSecondsDemo(nsec, nmsec)
{
    var d, s;
    var sep = ":";
    d = new Date();
    d.setUTCSeconds(nsec, nmsec);
    s = "Current setting is ";
    s += d.toUTCString() + sep + d.getUTCMillis;
    return(s);
}
```
**toLocaleString Method**

**Description**

Returns a date converted to a string using the current locale.

**Syntax**

`dateObj.toLocaleString()`

**Remarks**

The `toLocaleString` method returns a `String` object that contains the date written in the current locale's default format. The format of the return value depends on the current locale. For example, in the United States, `toLocaleString` may return "01/05/96 00:00:00" for January 5, but in Europe, it may return "05/01/96 00:00:00" for the same date, as European convention puts the day before the month.

The following example illustrates the use of the `toLocaleString` method:

```javascript
function toLocaleStrDemo() {
    var d, s;
    d = new Date();
}  ```
s = "Current setting is ";
s += d.toLocaleString();
return(s);
}
Parse Method

See Also Applies To

Description

Parses a string containing a date, and returns the number of milliseconds between that date and midnight, January 1, 1970.

Syntax

Date.parse(dateVal)

The dateVal argument is either a string containing a date in a format such as "Jan 5, 1996 08:47:00" or a VT_DATE value retrieved from an ActiveX® object or other object.

Remarks

The parse method returns an integer value representing the number of milliseconds between midnight, January 1, 1970 and the date supplied in dateVal.

The parse method is a static method of the Date object. Because it is a static method, it is invoked as shown in the following example rather than invoked as a method of a created Date object.

```javascript
var datestring = "November 1, 1997 10:15 AM";
Date.parse(datestring)
```

The following rules govern what the parse method can successfully parse:

- Short dates can use either a "/" or "-" date separator, but must
follow the month/day/year format, for example "7/20/96".

- Long dates of the form "July 10 1995" can be given with the year, month, and day in any order, and the year in 2-digit or 4-digit form. If you use the 2-digit form, the year must be greater than or equal to 70.

- Any text inside parentheses is treated as a comment. These parentheses may be nested.

- Both commas and spaces are treated as delimiters. Multiple delimiters are permitted.

- Month and day names must have two or more characters. Two character names that are not unique are resolved as the last match. For example, "Ju" is resolved as July, not June.

- The stated day of the week is ignored if it is incorrect given the remainder of the supplied date. For example, "Tuesday November 9 1996" is accepted and parsed even though that date actually falls on a Friday. The resulting Date object contains "Friday November 9 1996".

- JScript handles all standard time zones, as well as Universal Coordinated Time (UTC) and Greenwich Mean Time (GMT).

- Hours, minutes, and seconds are separated by colons, although all need not be specified. "10:", "10:11", and "10:11:12" are all valid.

- If the 24-hour clock is used, it is an error to specify "PM" for times later than 12 noon. For example, "23:15 PM" is an error.

- A string containing an invalid date is an error. For example, a string containing two years or two months is an error.

The following example illustrates the use of the parse method:
function GetTimeTest(testdate) {
    var d, s, t;
    var MinMilli = 1000 * 60;
    var HrMilli = MinMilli * 60;
    var DyMilli = HrMilli * 24;
    d = new Date();
    t = Date.parse(testdate);
    s = "There are "
    s += Math.round(Math.abs(t / DyMilli)) + " day
s += "between " + testdate + " and 1/1/70";
    return(s);
}
Description

Returns the number of milliseconds between midnight, January 1, 1970 Universal Coordinated Time (UTC) (or GMT) and the supplied date.

Syntax

**Date.UTC(year, month, day[, hours[, minutes[, seconds[, ms]]]])**

The **UTC** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>Required. The full year designation is required for cross-century date accuracy. If year is between 0 and 99 is used, then year is assumed to be 1900 + year.</td>
</tr>
<tr>
<td>month</td>
<td>Required. The month as an integer between 0 and 11 (January to December).</td>
</tr>
<tr>
<td>date</td>
<td>Required. The date as an integer between 1 and 31.</td>
</tr>
<tr>
<td>hours</td>
<td>Optional. Must be supplied if minutes is supplied. An integer from 0 to 23 (midnight to 11pm) that specifies the</td>
</tr>
</tbody>
</table>
### Remarks

The **UTC** method returns the number of milliseconds between midnight, January 1, 1970 UTC and the supplied date. This return value can be used in the `setTime` method and in the **Date** object constructor. If the value of an argument is greater than its range or is a negative number, other stored values are modified accordingly. For example, if you specify 150 seconds, JScript redefines that number as two minutes and 30 seconds.

The difference between the **UTC** method and the **Date** object constructor that accepts a date is that the **UTC** method assumes UTC, and the **Date** object constructor assumes local time.

The **UTC** method is a static method. Therefore, a **Date** object does not have to be created before it can be used. The **UTC** method is invoked as follows:

```
var datestring = "November 1, 1997 10:15 AM";
Date.UTC(datestring)
```

---

**Note** If `year` is between 0 and 99, use **1900** + `year` for the year.

---

<table>
<thead>
<tr>
<th><strong>Field</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>hour</strong></td>
<td>Optional. Must be supplied if <strong>seconds</strong> is supplied. An integer from 0 to 59 that specifies the minutes.</td>
</tr>
<tr>
<td><strong>minutes</strong></td>
<td>Optional. Must be supplied if <strong>seconds</strong> is supplied. An integer from 0 to 59 that specifies the minutes.</td>
</tr>
<tr>
<td><strong>seconds</strong></td>
<td>Optional. Must be supplied if <strong>milliseconds</strong> is supplied. An integer from 0 to 59 that specifies the seconds.</td>
</tr>
<tr>
<td><strong>milliseconds</strong></td>
<td>Optional. An integer from 0 to 999 that specifies the milliseconds.</td>
</tr>
</tbody>
</table>
function DaysBetweenDateAndNow(yr, mo, dy) {
    var d, r, t1, t2, t3;
    var MinMilli = 1000 * 60
    var HrMilli = MinMilli * 60
    var DyMilli = HrMilli * 24
    t1 = Date.UTC(yr, mo, dy)
    d = new Date();
    t2 = d.getTime();
    if (t2 >= t1)
        t3 = t2 - t1;
    else
        t3 = t1 - t2;
    r = Math.round(t3 / DyMilli);
    return(r);
}
function Statement

See Also

Description

Declares a new function.

Syntax

function functionname([argument1 [, argument2 [, ...
...argumentn]])
{
    statements
}

The function statement syntax has the following parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>functionname</td>
<td>The name of the function.</td>
</tr>
<tr>
<td>argument1...argumentn</td>
<td>An optional, comma-separated list of arguments the function understands.</td>
</tr>
<tr>
<td>statements</td>
<td>One or more JScript statements.</td>
</tr>
</tbody>
</table>

Remarks
Use the `function` statement to declare a function for later use. The code contained in `statements` is not executed until the function is called from elsewhere in the script.

The following example illustrates the use of the `function` statement:

```javascript
function myfunction(arg1, arg2)
{
    var r;
    r = arg1 * arg2;
    return(r);
}
```

**Note** When calling a function, ensure that you always include the parentheses and any required arguments. Calling a function without parentheses causes the text of the function to be returned instead of the results of the function.
Microsoft® JScript®

**parseInt Method**

**See Also**

**Applies To**

**Description**

Returns an integer converted from a string.

**Syntax**

`parseInt(numstring, [radix])`

The `parseInt` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>numstring</code></td>
<td>Required. A string to convert into a number.</td>
</tr>
<tr>
<td><code>radix</code></td>
<td>Optional. A value between 2 and 36 indicating the base of the number contained in <code>numstring</code>. If not supplied, strings with a prefix of '0x' are considered hexadecimal and strings with a prefix of '0' are considered octal. All other strings are considered decimal.</td>
</tr>
</tbody>
</table>

**Remarks**

The `parseInt` method returns an integer value equal to the
number contained in `numstring`. If no prefix of `numstring` can be successfully parsed into an integer, `NaN` (not a number) is returned.

```javascript
parseInt("abc")    // Returns NaN.
parseInt("12abc")  // Returns 12.
```

You can test for `NaN` using the `isNaN` method.
**parseFloat Method**

**Description**

Returns a floating-point number converted from a string.

**Syntax**

```javascript
parseFloat(numstring)
```

The `numstring` argument is a string that contains a floating-point number.

**Remarks**

The `parseFloat` method returns an numerical value equal to the number contained in `numstring`. If no prefix of `numstring` can be successfully parsed into a floating-point number, `NaN` (not a number) is returned.

```javascript
parseFloat("abc")   // Returns NaN.
parseFloat("1.2abc") // Returns 1.2.
```

You can test for `NaN` using the `isNaN` method.
% Operator

See Also

Description

Divides two numbers and returns the remainder.

Syntax

\[ result = number1 \% number2 \]

The % operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any variable.</td>
</tr>
<tr>
<td>number1</td>
<td>Any numeric expression.</td>
</tr>
<tr>
<td>number2</td>
<td>Any numeric expression.</td>
</tr>
</tbody>
</table>

Remarks

The modulus, or remainder, operator divides \( number1 \) by \( number2 \) (rounding floating-point numbers to integers) and returns only the remainder as \( result \). For example, in the following expression, \( A \) (which is \( result \)) equals 5.

\[ A = 19 \% 6.7 \]

For information on when a run-time error is generated by the % operator, see the Operator Behavior table.
**Description**

Multiplies two numbers.

**Syntax**

\[ result = number1*number2 \]

The * operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any variable.</td>
</tr>
<tr>
<td>number1</td>
<td>Any expression.</td>
</tr>
<tr>
<td>number2</td>
<td>Any expression.</td>
</tr>
</tbody>
</table>

**Remarks**

For information on when a run-time error is generated by the * operator, see the [Operator Behavior](#) table.
See Also

Description

Used to divide two numbers and return a numeric result.

Syntax

\[ result = number1 / number2 \]

The `/` operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any numeric variable.</td>
</tr>
<tr>
<td>number1</td>
<td>Any numeric expression.</td>
</tr>
<tr>
<td>number2</td>
<td>Any numeric expression.</td>
</tr>
</tbody>
</table>

Remarks

For information on when a run-time error is generated by the `/` operator, see the Operator Behavior table.
Global Object

See Also  Methods  Properties

Description

An intrinsic object whose purpose is to collect global methods into one object.

Syntax

The **Global** object has no syntax. You call its methods directly.

Remarks

The **Global** object is never used directly, and cannot be created using the **new** operator. It is created when the scripting engine is initialized, thus making its methods and properties available immediately.
**Microsoft® JScript® split Method**

**Description**

Returns the array of strings that results when a string is separated into substrings.

**Syntax**

```javascript
stringObj.split(str)
```

The `split` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stringObj</code></td>
<td>Required. The <code>String</code> object or literal to be split. This object is not modified by the <code>split</code> method.</td>
</tr>
<tr>
<td><code>str</code></td>
<td>Required. A string or <code>Regular Expression</code> object describing what character is used to define where the splits take place.</td>
</tr>
</tbody>
</table>

**Remarks**

The result of the `split` method is an array of strings split at each point where `str` occurred in `stringObj`.

The following example illustrates the use of the `split` method:

```javascript
function SplitDemo()
```
function splitSentence(sentence) {
    var s, ss;
    var s = "The quick brown fox jumped over the
towards the lazy yellow dog."
    // Split at each space character.
    ss = s.split(" ");
    return(ss);
}
**getVarDate** Method

**See Also**

** Applies To **

---

**Description**

Returns the VT_DATE value in a **Date** object.

**Syntax**

```
    dateobj.getVarDate( )
```

The `dateobj` argument is any **Date** object.

**Remarks**

The **getVarDate** method is used when interacting with ActiveX® objects or other objects that accept and return date values in VT_DATE format.
Description

Returns, as an array, the results of a search on a string using a supplied *Regular Expression* object.

Syntax

```
stringObj.match(rgExp)
```

The `match` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stringObj</code></td>
<td>Required. The <em>String</em> object or literal on which to perform the search.</td>
</tr>
<tr>
<td><code>rgExp</code></td>
<td>Required. The regular expression to use in the search.</td>
</tr>
</tbody>
</table>

Remarks

The `match` method, which behaves like the `exec` method, returns an array of values. Element zero of the array contains the last matched characters. Elements 1...n contain matches to any parenthesized substrings in the regular expression.

The method updates the contents of the *RegExp* object.

The following example illustrates the use of the `match` method:
function MatchDemo()
{
    var r, re;
    var s = "The quick brown fox jumped over thee
    re = /fox/i;
    r = s.match(re);
    return(r);
}

replace Method

Description

Returns a copy of a string with text replaced using a regular expression.

Syntax

```
stringObj.replace(rgExp, replaceText)
```

The `replace` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stringObj</code></td>
<td>Required. The <code>String</code> object or literal on which to perform the replace. This object is not modified by the <code>replace</code> method.</td>
</tr>
<tr>
<td><code>rgExp</code></td>
<td>Required. A <code>Regular Expression</code> object describing what to search for.</td>
</tr>
<tr>
<td><code>replaceText</code></td>
<td>Required. A <code>String</code> object or literal containing the text to replace for every successful match of <code>rgExp</code> in <code>stringObj</code>.</td>
</tr>
</tbody>
</table>

Remarks

The result of the `replace` method is a copy of `stringObj` after all replacements have been made.

The method updates the contents of the `RegExp` object.
The following example illustrates the use of the `replace` method:

```javascript
function ReplaceDemo()
{
    var r, re;
    var s = "The quick brown fox jumped over the
to的数据");
    return r;
}
```

In addition, the `replace` method can also replace subexpressions in the pattern. The following example swaps each pair of words in the string:

```javascript
function ReplaceDemo()
{
    var r, re;
    var s = "The quick brown fox jumped over the
to的数据");
    return r;
}
```
OpenAsTextStream Method

Description

Opens a specified file and returns a TextStream object that can be used to read from, write to, or append to the file.

Syntax

object.OpenAsTextStream([iomode, [format]])

The OpenAsTextStream method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a File object.</td>
</tr>
<tr>
<td>iomode</td>
<td>Optional. Indicates input/output mode. Can be one of three constants: ForReading, ForWriting, or ForAppending.</td>
</tr>
<tr>
<td>format</td>
<td>Optional. One of three Tristate values used to indicate the format of the opened file. If omitted, the file is opened as ASCII.</td>
</tr>
</tbody>
</table>

Settings

The iomode argument can have any of the following settings:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>
**ForReading** 1 Open a file for reading only. You can’t write to this file.

**ForWriting** 2 Open a file for writing. If a file with the same name exists, its previous contents are overwritten.

**ForAppending** 8 Open a file and write to the end of the file.

The *format* argument can have any of the following settings:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TristateUseDefault</td>
<td>-2</td>
<td>Opens the file using the system default.</td>
</tr>
<tr>
<td>TristateTrue</td>
<td>-1</td>
<td>Opens the file as Unicode.</td>
</tr>
<tr>
<td>TristateFalse</td>
<td>0</td>
<td>Opens the file as ASCII.</td>
</tr>
</tbody>
</table>

**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.

The following code illustrates the use of the **OpenAsTextStream** method:

```javascript
function TextStreamTest() {
    var fso, f, ts, s;
    var ForReading = 1, ForWriting = 2, ForApp
    var TristateUseDefault = -2, TristateTrue = -1,
    fso = new ActiveXObject("Scripting.FileSystemObject");
```
fso.CreateTextFile( "test1.txt" );     // Create
f = fso.GetFile("test1.txt");
ts = f.OpenAsTextStream(ForWriting, TristateUseDefault);ts.Write( "Hello World" );
ts.Close( );
ts = f.OpenAsTextStream(ForReading, TristateUseDefault);s = ts.ReadLine( );ts.Close( );return(s);
}
Microsoft® JScript® abs Method

Applies To

Math Object
Microsoft® JScript® acos Method

Applies To

Math Object
ActiveXObject

Object

See Also

GetObject Function
The following table describes the behavior of most Microsoft JScript operators. The columns and rows represent the different types of expressions possible on either side of an operator in JScript, and the entries in the table describe the behavior.

An E indicates a run-time error. An N indicates a numeric result, or a Boolean result in the case of logical operators.

<table>
<thead>
<tr>
<th></th>
<th>obj</th>
<th>as</th>
<th>ns</th>
<th>num</th>
<th>bool</th>
<th>undef</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>N</td>
<td>E</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>as</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>ns</td>
<td>N</td>
<td>E</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>num</td>
<td>N</td>
<td>E</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>bool</td>
<td>N</td>
<td>E</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>undef</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>null</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
</tbody>
</table>

obj = object, as = alphanumeric string, ns = numeric string, num = number, bool = Boolean, undef = undefined, null = null value.
Microsoft® JScript® anchor Method

See Also

link Method
String Object Methods
String Object Properties
Microsoft® JScript® anchor Method

Applies To

String Object
Microsoft® JScript® **asin Method**

Applies To

**Math Object**
Microsoft® JScript® atan Method

Applies To

Math Object

Language Reference
Microsoft® JScript® atan2 Method

See Also

atan Method
Math Object Methods
tan Method
Microsoft® JScript® atan2 Method

Applies To

Math Object
Microsoft® JScript®  

**atEnd Method**

See Also

- `item Method`
- `moveFirst Method`
- `moveNext Method`
Microsoft® JScript® atEnd Method

Applies To

Enumerator Object
Microsoft® JScript® big Method

See Also

small Method
String Object Methods
String Object Properties
Microsoft® JScript® big Method

Applies To

String Object
Microsoft® JScript® & Operator

See Also

&= Operator
Operator Behavior
Operator Precedence
Operator Summary
<< Operator

See Also

<<= Operator
>> Operator
>>> Operator
Operator Behavior
Operator Precedence
Operator Summary
See Also

! Operator
Operator Behavior
Operator Precedence
Operator Summary
See Also

|= Operator
Operator Behavior
Operator Precedence
Operator Summary
See Also

<< Operator
>>= Operator
>>> Operator
Operator Behavior
Operator Precedence
Operator Summary
See Also

String Object Methods
String Object Properties
Microsoft® JScript® blink Method

Applies To

String Object
Microsoft® JScript® **bold Method**

See Also

*italics Method*

**String Object Methods**

**String Object Properties**
Microsoft® JScript® **bold Method**

Applies To

---

**String Object**
Boolean Object

See Also

new Operator
var Statement
Members of Boolean.prototype

toString Method
valueOf Method

Nonmembers of Boolean.prototype

The Boolean object has no methods that are not part of the prototype.
Microsoft® JScript® Boolean Object

Properties

Members of Boolean.prototype

constructor Property

Nonmembers of Boolean.prototype

prototype Property
Microsoft® JScript® break Statement

See Also

continue Statement
do...while Statement
for Statement
for...in Statement
Labeled Statement
while Statement
Microsoft® JScript®

@cc_on

Statement

See Also

Conditional Compilation
Conditional Compilation Variables
@if Statement
@set Statement
Microsoft® JScript® ceil Method

Applies To

Math Object

Language Reference
Microsoft® JScript® charAt Method

See Also

String Object Methods
String Object Properties
Microsoft® JScript® charAt Method

Applies To

String Object
Microsoft® JScript® charCodeAt Method

See Also

fromCharCode Method
String Object Methods
Microsoft® JScript® charCodeAt Method Applies To

String Object
Microsoft® JScript®, Operator

See Also

for Statement
Operator Behavior
Operator Precedence
Operator Summary
Comparison Operators

See Also

Operator Behavior
Operator Precedence
Operator Summary
Microsoft® JScript®

**compile Method**

**See Also**

- [Regular Expression Object Methods](#)
- [Regular Expression Object Properties](#)
- [Regular Expression Syntax](#)
Microsoft® JScript® compile Method

Applies To

Regular Expression Object
**+= Operator**

**See Also**

**Description**

Used to increment a variable by a specified amount.

**Syntax**

\[ result += expression \]

The `+=` operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>result</code></td>
<td>Any variable</td>
</tr>
<tr>
<td><code>expression</code></td>
<td>Any expression</td>
</tr>
</tbody>
</table>

**Remarks**

Using this operator is exactly the same as specifying:

\[ result = result + expression \]

The underlying subtype of the expressions determines the behavior of the `+=` operator.

<table>
<thead>
<tr>
<th>If</th>
<th>Then</th>
</tr>
</thead>
<tbody>
<tr>
<td>Both expressions are numeric or Boolean</td>
<td>Add.</td>
</tr>
<tr>
<td>Both expressions are strings</td>
<td>Concatenate.</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>One expression is numeric and the other is a string</td>
<td>Concatenate.</td>
</tr>
</tbody>
</table>

For information on when a run-time error is generated by the += operator, see the Operator Behavior table.
See Also

Description

Used to perform a bitwise AND on an expression.

Syntax

\[ \text{result} \&= \text{expression} \]

The &\text{=} operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any variable</td>
</tr>
<tr>
<td>expression</td>
<td>Any expression</td>
</tr>
</tbody>
</table>

Remarks

Using this operator is exactly the same as specifying:

\[ \text{result} = \text{result} \& \text{expression} \]

The &\text{=} operator looks at the binary representation of the values of result and expression and does a bitwise AND operation on them. The output of this operation behaves like this:

\[
\begin{align*}
0101 & \quad \text{(result)} \\
1100 & \quad \text{(expression)}
\end{align*}
\]
0100   (output)

Any time both of the expressions have a 1 in a digit, the result has a 1 in that digit. Otherwise, the result has a 0 in that digit.

For information on when a run-time error is generated by the & operator, see the Operator Behavior table.
Microsoft® JScript® | = Operator

See Also

Description

Used to perform a bitwise OR on an expression.

Syntax

\[ \text{result} \mid = \text{expression} \]

The | = operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any variable.</td>
</tr>
<tr>
<td>expression</td>
<td>Any expression.</td>
</tr>
</tbody>
</table>

Remarks

Using this operator is exactly the same as specifying:

\[ \text{result} = \text{result} \mid \text{expression} \]

The | = operator looks at the binary representation of the values of \text{result} and \text{expression} and does a bitwise OR operation on them. The result of this operation behaves like this:

\[
\begin{align*}
0101 & \quad (\text{result}) \\
1100 & \quad (\text{expression})
\end{align*}
\]
1101 (output)

Any time either of the expressions has a 1 in a digit, the result has a 1 in that digit. Otherwise, the result has a 0 in that digit.

For information on when a run-time error is generated by the |= operator, see the Operator Behavior table.
**Description**

Used to perform a bitwise exclusive OR on an expression.

**Syntax**

\[ \text{result} \^= \text{expression} \]

The ^= operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any variable</td>
</tr>
<tr>
<td>expression</td>
<td>Any expression</td>
</tr>
</tbody>
</table>

**Remarks**

Using the ^= operator is exactly the same as specifying:

\[ \text{result} = \text{result} \^ \text{expression} \]

The ^= operator looks at the binary representation of the values of two expressions and does a bitwise exclusive OR operation on them. The result of this operation behaves as follows:

\[
\begin{align*}
0101 & \quad \text{(result)} \\
1100 & \quad \text{(expression)}
\end{align*}
\]
When one, and only one, of the expressions has a 1 in a digit, the result has a 1 in that digit. Otherwise, the result has a 0 in that digit.

For information on when a run-time error is generated by the ^= operator, see the Operator Behavior table.
Microsoft® JScript® \(/=\) Operator

See Also

Description

Used to divide a variable by an expression.

Syntax

\( result /= expression \)

The \( /= \) operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( result )</td>
<td>Any numeric variable.</td>
</tr>
<tr>
<td>( expression )</td>
<td>Any numeric expression.</td>
</tr>
</tbody>
</table>

Remarks

Using the \( /= \) operator is exactly the same as specifying:

\[ result = result / expression \]

For information on when a run-time error is generated by the \( /= \) operator, see the Operator Behavior table.
Description

Used to shift the bits of an expression to the left.

Syntax

\[ result <<= expression \]

The <<= operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any variable</td>
</tr>
<tr>
<td>expression</td>
<td>Any expression</td>
</tr>
</tbody>
</table>

Remarks

Using the <<= operator is exactly the same as specifying:

\[ result = result <<= expression \]

The <<= operator shifts the bits of result left by the number of bits specified in expression. For example:

\[ var temp \]
temp = 14
temp <<= 2

The variable temp has a value of 56 because 14 (00001110 in binary) shifted left two bits equals 56 (00111000 in binary). Bits are filled in with zeroes when shifting.

For information on when a run-time error is generated by the <<= operator, see the Operator Behavior table.
See Also

Description

Used to divide two numbers and return only the remainder.

Syntax

\[ result \%= expression \]

The %= operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any variable.</td>
</tr>
<tr>
<td>expression</td>
<td>Any numeric expression.</td>
</tr>
</tbody>
</table>

Remarks

Using the %= operator is exactly the same as specifying:

\[ result = result \% expression \]

For information on when a run-time error is generated by the %= operator, see the Operator Behavior table.
** *= Operator

See Also

Description

Used to multiply a number by another number.

Syntax

\[ result *= expression \]

The *= operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any variable.</td>
</tr>
<tr>
<td>expression</td>
<td>Any expression.</td>
</tr>
</tbody>
</table>

Remarks

Using the *= operator is exactly the same as specifying:

\[ result = result * expression \]

For information on when a run-time error is generated by the *= operator, see the Operator Behavior table.
See Also

Description

Used to shift the bits of an expression to the right, preserving sign.

Syntax

\[ result \gg= expression \]

The \( \gg= \) operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any variable.</td>
</tr>
<tr>
<td>expression</td>
<td>Any expression.</td>
</tr>
</tbody>
</table>

Remarks

Using the \( \gg= \) operator is exactly the same as specifying:

\[ result = result \gg expression \]

The \( \gg= \) operator shifts the bits of \( result \) right by the number of bits specified in \( expression \). The sign bit of \( result \) is used to fill the digits from the left. Digits shifted off the right are discarded. For example, after the following code is evaluated, \( temp \) has a value of -4: 14 (11110010 in binary) shifted right two bits equals -4 (11111100 in binary).
var temp

temp = -14

temp >>= 2

For information on when a run-time error is generated by the >>= operator, see the Operator Behavior table.
See Also

Description

Used to subtract the value of an expression from a variable.

Syntax

\[ result -= expression \]

The -= operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any numeric variable.</td>
</tr>
<tr>
<td>expression</td>
<td>Any numeric expression.</td>
</tr>
</tbody>
</table>

Remarks

Using the -= operator is exactly the same as doing the following:

\[ result = result - expression \]

For information on when a run-time error is generated by the - operator, see the Operator Behavior table.
### See Also

### Description

Used to make an unsigned right shift of the bits in a variable.

### Syntax

\[
result >>>= expression
\]

The \( \gggg = \) operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any variable.</td>
</tr>
<tr>
<td>expression</td>
<td>Any expression.</td>
</tr>
</tbody>
</table>

### Remarks

Using the \( \gggg = \) operator is exactly the same as doing the following:

\[
result = result >>> expression
\]

The \( \gggg = \) operator shifts the bits of \( result \) right by the number of bits specified in \( expression \). Zeroes are filled in from the left. Digits shifted off the right are discarded. For example:

\[
var temp
\]
temp = -14

temp >>= 2

The variable temp has a value of 1073741820 as -14 (11111111 1111111 11111111 11110010 in binary) shifted right two bits equals 1073741820 (00111111 11111111 11111111 11111100 in binary).

For information on when a run-time error is generated by the >>= operator, see the Operator Behavior table.
Microsoft® JScript® **concat** Method

**(Array)**

See Also

**concat Method (String)**
**join Method**
Microsoft® JScript® `concat` Method

(Array)

Applies To

Array Object
String Object
Microsoft® JScript® **concat Method**

(String)

See Also

Addition Operator (+)
concat Method (Array)
String Object Methods
Microsoft® JScript® **concat Method**

(String)

Applies To

**Array Object**

**String Object**
Conditional Compilation

See Also

Conditional Compilation Variables
@cc_on Statement
@if Statement
@set Statement
Conditional Compilation

Variables

See Also

Conditional Compilation
@cc_on Statement
@if Statement
@set Statement
if...else Statement
Operator Behavior
Operator Precedence
Operator Summary
Microsoft® JScript® constructor Property

See Also

prototype Property
Property
Applies To

Array Object
Boolean Object
Date Object
Function Object
Math Object
Number Object
Object Object
String Object
continue

Statement

See Also

break Statement
do...while Statement
for Statement
for...in Statement
Labeled Statement
while Statement
Microsoft® JScript® cos Method

Applies To

Math Object
See Also

new Operator
t
var Statement

Language Reference
Microsoft® JScript® Date Object

Properties

constructor Property
prototype Property
Operators

See Also

- Operator Behavior
- Operator Precedence
- Operator Summary
Microsoft® JScript®

description

Property

See Also

number Property
Error Object
dimensions

Method

See Also

getItem Method
lbound Method
toArray Method
ubound Method
Method

Applies To

VBArray Object
do...while Statement

See Also

break Statement
continue Statement
for Statement
for...in Statement
while Statement
Labeled Statement
Property
Applies To

Math Object
Microsoft® JScript® Enumerator

Object

See Also

Drives Collection
Files Collection
Folders Collection
Microsoft® JScript® Enumerator

Object Methods

atEnd Method
item Method
moveFirst Method
moveNext Method
The **Enumerator** object has no properties.
Microsoft® JScript® Error Object

See Also

new Operator
throw Statement
try...catch Statement
var Statement
Microsoft® JScript® Error Object

Properties

description Property
number Property
Microsoft® JScript® escape Method

See Also

String Object
unescape Method
Microsoft® JScript® escape Method

Applies To

Global Object
Microsoft® JScript® eval Method

See Also

String Object
Microsoft® JScript® **eval Method**

**Applies To**

**Global Object**
Microsoft® JScript® exec Method

See Also

RegExp Object
Regular Expression Object Methods
Regular Expression Object Properties
Regular Expression Syntax
Microsoft® JScript® exec Method

Applies To

Regular Expression Object
Microsoft® JScript® **exp Method**

Applies To

---

**Math Object**
See Also

String Object Methods
String Object Properties
**Microsoft® JScript®**

**fixed Method**

**Applies To**

---

**String Object**
Microsoft® JScript® floor Method

Applies To

Math Object
See Also

fontsize Method
String Object Methods
String Object Properties
Microsoft® JScript®  

fontcolor Method

Applies To

String Object
Microsoft® JScript® fontsize Method

See Also

tfontcolor Method
String Object Methods
String Object Properties

Language Reference
Microsoft® JScript® \texttt{fontsize} Method

Applies To

\texttt{String Object}
See Also

for...in Statement
while Statement
Microsoft® JScript®

for...in Statement

See Also

for Statement
while Statement
Microsoft® JScript® `fromCharCode`

Method

See Also

`charCodeAt Method`

`String Object Methods`
Microsoft® JScript® `fromCharCode`

**Method**

** Applies To **

**String Object**

[Language Reference]
See Also

function Statement
new Operator
var Statement
Members of Function.prototype

toString Method
valueOf Method

Nonmembers of Function.prototype

The Function object has no methods that are not part of the prototype.
Microsoft® JScript® Function Object

Properties

Members of Function.prototype

arguments Property
caller Property

constructor Property

Nonmembers of Function.prototype

prototype Property
Microsoft® JScript® getItem Method

See Also

dimensions Method
lbound Method
toArray Method
ubound Method
Microsoft® JScript® getItem Method

Applies To

VBArray Object
**GetObject**

Function

See Also

[ActiveXObject Object](#)
See Also

Conditional Compilation
Conditional Compilation Variables
@cc_on Statement
@set Statement
Microsoft® JScript® if...else Statement

See Also

Conditional Operator (?:)
Microsoft® JScript® **indexOf Method**

**Applies To**

**String Object**
Microsoft® JScript® isNaN Method

See Also

isFinite Method
NaN Property (Global)
parseFloat Method
parseInt Method
Microsoft® JScript® isnaN Method

Applies To

Global Object
Microsoft® JScript® *italics Method*

**See Also**

- **bold Method**
- **String Object Methods**
- **String Object Properties**
Microsoft® JScript® **italics Method**

Applies To

**String Object**
See Also

atEnd Method
moveFirst Method
moveNext Method
item Method
Applies To

Enumerator Object
Microsoft® JScript® join Method

See Also

Array Object Methods
String Object
Microsoft® JScript® join Method
Applies To

Array Object
Microsoft® JScript®

Labeled Statement

See Also

break Statement
continue Statement
Microsoft® JScript®

lastIndexOf

Method

See Also

indexOf Method
String Object Methods
String Object Properties
Microsoft® JScript® lastIndexOf Method

Applies To

String Object
Microsoft® JScript® lbound Method

See Also

dimensions Method
getItem Method
toArray Method
ubound Method
Microsoft® JScript® `Ibound Method`

Applies To

VBArray Object
length Property

(Array)

See Also

length Property (Function)
length Property (String)
Microsoft® JScript® `length` Property

(Array)
Applies To

Array Object
Microsoft® JScript® length Property

(Function)

See Also

arguments Property
length Property (Array)
length Property (String)
Microsoft® JScript® length Property
(String)
See Also

length Property (Array)
length Property (Function)
String Object Methods
String Object Properties
Microsoft® JScript® length Property

(String)

Applies To

String Object
link Method

See Also

anchor Method
String Object Methods
String Object Properties
Microsoft® JScript® link Method

Applies To

String Object
Microsoft® JScript® LN2 Property

Applies To

Math Object
Microsoft® JScript®  LN10 Property
Applies To

Math Object
Microsoft® JScript® log Method

Applies To

Math Object
Microsoft® JScript® LOG2E Property

Applies To

Math Object
Microsoft® JScript® Log10E Property

Applies To

Math Object
Microsoft® JScript® Math Object

See Also

Number Object
Microsoft® JScript® max Method

Applies To

Math Object
Microsoft® JScript® **min Method**

Applies To

**Math Object**
Microsoft® JScript® moveFirst Method

See Also

atEnd Method
item Method
moveNext Method
Microsoft® JScript® moveFirst Method Applies To Enumerator Object
moveNext Method

See Also

atEnd Method
item Method
moveFirst Method
Microsoft® JScript® moveNext Method
Applies To Enumerator Object
Microsoft® JScript® NaN Property

Applies To

Number Object
See Also

function Statement
Microsoft® JScript® number Property

See Also

description Property
Microsoft® JScript® number Property

Applies To

Error Object
Microsoft® JScript® Object Object

See Also

Function Object
Global Object
Members of Object.prototype

**toString Method**
**valueOf Method**

Non-members of Object.prototype

The **Object** object has no methods that are not part of the prototype.
Members of Object.prototype

prototype Property
constructor Property

Non-members of Object.prototype

The **Object** object has no properties that are not part of the prototype.
Microsoft® JScript®  **PI Property**  
Applies To  

**Math Object**
Microsoft® JScript® *pow Method*

Applies To

**Math Object**
prototype

Property

See Also

constructor Property
**Microsoft® JScript® prototype**

**Property**

Applies To

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Array Object</strong></td>
</tr>
<tr>
<td><strong>Boolean Object</strong></td>
</tr>
<tr>
<td><strong>Date Object</strong></td>
</tr>
<tr>
<td><strong>Function Object</strong></td>
</tr>
<tr>
<td><strong>Number Object</strong></td>
</tr>
<tr>
<td><strong>Object Object</strong></td>
</tr>
<tr>
<td><strong>String Object</strong></td>
</tr>
</tbody>
</table>
Microsoft® JScript® random Method

Applies To

Math Object
See Also

function Statement
Microsoft® JScript® reverse Method

See Also

Array Object Methods
Microsoft® JScript® reverse Method

Applies To

Array Object
Microsoft® JScript® round Method

Applies To

Math Object
Microsoft® JScript® ScriptEngine Function

See Also

ScriptEngineBuildVersion Function
ScriptEngineMajorVersion Function
ScriptEngineMinorVersion Function
ScriptEngineBuildVersion Function

See Also

ScriptEngine Function
ScriptEngineMajorVersion Function
ScriptEngineMinorVersion Function
ScriptEngineMajorVersion Function

See Also

ScriptEngine Function
ScriptEngineBuildVersion Function
ScriptEngineMinorVersion Function
ScriptEngineMinorVersion Function

See Also

ScriptEngine Function
ScriptEngineBuildVersion Function
ScriptEngineMajorVersion Function
Microsoft® JScript® *sin* Method

 Applies To

**Math Object**
See Also

slice Method (String)
Microsoft® JScript® slice Method

Applies To

Array Object
String Object
Microsoft® JScript® slice Method
(String)
See Also

slice Method (Array)
String Object Methods

Language Reference
Microsoft® JScript® slice Method
(String)
Applies To

Array Object
String Object
See Also

big Method
String Object Methods
String Object Properties
Microsoft® JScript® **small Method**

Applies To

**String Object**
Microsoft® JScript® sort Method

See Also

Array Object Methods
Microsoft® JScript® sort Method

Applies To

Array Object
See Also

- Regular Expression Object Methods
- Regular Expression Object Properties
- Regular Expression Syntax
Microsoft® JScript® source Property

Applies To

Regular Expression Object
Microsoft® JScript® sqrt Method

Applies To

Math Object
Microsoft® JScript® SQRT1_2 Property
Applies To

Math Object
Microsoft® JScript® SQRT2 Property

Applies To

Math Object
strike Method

See Also

String Object Methods
String Object Properties
Microsoft® JScript® strike Method

Applies To

String Object
Microsoft® JScript® String Object

See Also

new Operator
Microsoft® JScript® String Object

Properties

Members of String.prototype

constructor Property

Nonmembers of String.prototype

length Property
prototype Property
Microsoft® JScript® sub Method

See Also

String Object Methods
String Object Properties
sup Method
Microsoft® JScript® sub Method

Applies To

String Object
Microsoft® JScript® \textbf{substr Method}

\section*{See Also}

\begin{itemize}
  \item \texttt{String Object Methods}
  \item \texttt{String Object Properties}
  \item \texttt{substring Method}
\end{itemize}
Microsoft® JScript® `substr` Method

Applies To

String Object
Microsoft® JScript® substring Method

See Also

String Object Methods
String Object Properties
substr Method
Microsoft® JScript® substring Method

Applies To

String Object
Microsoft® JScript® sup Method

See Also

String Object Methods
String Object Properties
sub Method
Microsoft® JScript® sup Method

Applies To

String Object
Microsoft® JScript®

switch Statement

See Also

break Statement
if...else Statement

Language Reference
Microsoft® JScript® \textbf{tan Method}

Applies To

\textbf{Math Object}
See Also

RegExp Object
Regular Expression Object Methods
Regular Expression Object Properties
Regular Expression Syntax
Microsoft® JScript® **test Method**

**Applies To**

**Regular Expression Object**
See Also

new Operator
See Also

dimensions Method
getItem Method
lbound Method
ubound Method
Microsoft® JScript® **toArray Method**

**Applies To**

**VBArray Object**
Microsoft® JScript® toLowerCase

Method

See Also

String Object Methods
String Object Properties
toUpperCase Method
Microsoft® JScript®

**toLowerCase**

**Method**

Applies To

---

**String Object**

---
Microsoft® JScript® \texttt{toString} Method

See Also

\texttt{function Statement}
### Microsoft® JScript® `toString` Method

Applies To

<table>
<thead>
<tr>
<th>Object Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array Object</td>
</tr>
<tr>
<td>Boolean Object</td>
</tr>
<tr>
<td>Function Object</td>
</tr>
<tr>
<td>Number Object</td>
</tr>
<tr>
<td>Object Object</td>
</tr>
<tr>
<td>String Object</td>
</tr>
</tbody>
</table>
Method

See Also

String Object Methods
String Object Properties
toLowerCase Method
Microsoft® JScript® toUpperCase

Method

Applies To

String Object

Language Reference
See Also

- Operator Behavior
- Operator Precedence
- Operator Summary
Microsoft® JScript®  ubound Method

See Also

dimensions Method
getItem Method
lbound Method
toArray Method
Microsoft® JScript® ubound Method

Applies To

VBArray Object
Microsoft® JScript® unescape Method

See Also

escape Method
String Object

Language Reference
Microsoft® JScript® unescape Method

Applies To

Global Object
See Also

>>> Operator
<< Operator
>> Operator
Operator Behavior
Operator Precedence
Operator Summary
Microsoft® JScript® valueOf Method

See Also

toString Method
Microsoft® JScript® **valueOf Method**

**Applies To**

- Array Object
- Boolean Object
- Date Object
- Function Object
- Number Object
- Object Object
- String Object
var Statement

See Also

function Statement
new Operator
Microsoft® JScript® VBArray Object

See Also

Array Object
VBArray Object

Methods

dimensions Method
getItem Method
lbound Method
toArray Method
ubound Method
The **VBAArray** object has no properties.
See Also

break Statement
continue Statement
do...while Statement
for Statement
for...in Statement
Microsoft® JScript® with Statement

See Also

this Statement
Microsoft® JScript® isFinite Method

See Also

isNaN Method
Microsoft® JScript® isFinite Method

Applies To

Global Object
search Method

See Also

exec Method
match Method
replace Method
String Object Methods
test Method
Microsoft® JScript® search Method

Applies To

String Object
See Also

Operator Behavior
Operator Precedence
Operator Summary
Microsoft® JScript® Add Method
(Dictionary)
See Also

Add Method (Folders)
Exists Method
Items Method
Keys Method
Remove Method
RemoveAll Method
Microsoft® JScript® Add Method
(Dictionary)
Applies To

Dictionary Object

Scripting Run-Time Reference
Microsoft® JScript® Add Method (Folders)

See Also

Add Method (Dictionary)
Microsoft® JScript® Add Method (Folders) Applies To

Folders Collection
AtEndOfLine Property
Applies To

TextStream Object
Microsoft® JScript® AtEndOfStream Property

See Also

AtEndOfLine Property
Microsoft® JScript® **AtEndOfStream**

**Property**

Applies To

---

**TextStream Object**
Attributes

Property

See Also

DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drive Property
Files Property
IsRootFolder Property
Name Property
ParentFolder Property
Path Property
ShortName Property
ShortPath Property
Size Property
SubFolders Property
Type Property
Microsoft® JScript® Attributes

Property Applies To

File Object
Folder Object
AvailableSpace Property

See Also

DriveLetter Property
DriveType Property
FileSystem Property
FreeSpace Property
IsReady Property
Path Property
RootFolder Property
SerialNumber Property
ShareName Property
TotalSize Property
VolumeName Property
Microsoft® JScript® AvailableSpace Property Applies To

Drive Object
Microsoft® JScript® BuildPath

Method

See Also

GetAbsolutePathName Method
GetBaseName Method
GetDriveName Method
GetExtensionName Method
GetFileName Method
GetParentFolderName Method
GetTempName Method
Microsoft® JScript® BuildPath Method Applies To FileSystemObject Object
Microsoft® JScript® Close Method

See Also

- Read Method
- Write Method

[Scripting Run-Time Reference]
Microsoft® JScript® Close Method

Applies To

TextStream Object
Microsoft® JScript®  

Column Property  

See Also  

Line Property  

Scripting Run-Time Reference
Microsoft® JScript® Column Property

Applies To

TextStream Object
See Also

CopyFile Method
CopyFolder Method
Delete Method
Move Method
OpenAsTextStream Method
Microsoft® JScript® Copy Method

Applies To

File Object
Folder Object
Microsoft® JScript®

CopyFile Method

See Also

Copy Method
CopyFolder Method
CreateTextFile Method
DeleteFile Method
MoveFile Method

Scripting Run-Time Reference
Microsoft® JScript® CopyFile Method

Applies To

FileSystemObject Object
Microsoft® JScript® CopyFolder Method

See Also

CopyFile Method
Copy Method
CreateFolder Method
DeleteFolder Method
MoveFolder Method
Microsoft® JScript® 

**CopyFolder**

**Method**

Applies To

**FileSystemObject** Object
See Also

CompareMode Property
Item Property
Key Property
Microsoft® JScript® Count Property

Applies To

Dictionary Object
Drives Collection
Files Collection
Folders Collection
Microsoft® JScript® **CreateFolder** Method

See Also

- **CopyFolder Method**
- **DeleteFolder Method**
- **MoveFolder Method**

[Scripting Run-Time Reference]
Microsoft® JScript® CreateFolder

Method
Applies To

FileSystemObject Object
Microsoft® JScript® 

CreateTextFile

Method

See Also

CreateFolder Method
OpenAsTextStream Method
OpenTextFile Method
Microsoft® JScript®

CreateTextFile

Method

Applies To

FileSystemObject Object
Folder Object
Microsoft® JScript®

DateCreated

Property

See Also

Attributes Property
DateLastAccessed Property
DateLastModified Property
Drive Property
Files Property
IsRootFolder Property
Name Property
ParentFolder Property
Path Property
ShortName Property
ShortPath Property
Size Property
SubFolders Property
Type Property
Microsoft® JScript® DateCreated

Property
Applies To

File Object
Folder Object
DateLastAccessed Property
See Also

Attributes Property
DateCreated Property
DateLastModified Property
Drive Property
Files Property
IsRootFolder Property
Name Property
ParentFolder Property
Path Property
ShortName Property
ShortPath Property
Size Property
SubFolders Property
Type Property
Microsoft® JScript®

DateLastAccessed Property
Applies To

File Object
Folder Object
DateLastModified Property

See Also

Attributes Property
DateCreated Property
DateLastAccessed Property
Drive Property
Files Property
IsRootFolder Property
Name Property
ParentFolder Property
Path Property
ShortName Property
ShortPath Property
Size Property
SubFolders Property
Type Property
DateLastModified

Property

Applies To

File Object
Folder Object
Delete Method

See Also

Copy Method
DeleteFile Method
DeleteFolder Method
Move Method
OpenAsTextStream Method
Microsoft® JScript® Delete Method

Applies To

File Object
Folder Object
**DeleteFile Method**

See Also

* CopyFile Method
* CreateTextFile Method
* Delete Method
* DeleteFolder Method
* MoveFile Method
Microsoft® JScript® **DeleteFile**

Method

Applies To

**FileSystemObject** Object
DeleteFolder Method

See Also

CopyFolder Method
CreateFolder Method
Delete Method
DeleteFile Method
MoveFolder Method
Microsoft® JScript® **DeleteFolder** Method

<table>
<thead>
<tr>
<th>Applies To</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileSystemObject Object</td>
</tr>
</tbody>
</table>
Dictionary Object Methods

- Add Method (Dictionary)
- Exists Method
- Items Method
- Keys Method
- Remove Method
- RemoveAll Method
Drive Object

See Also

Drives Collection
File Object
Files Collection
Folder Object
Folders Collection
GetDrive Method
Microsoft® JScript® Drive Object

Properties

AvailableSpace Property
DriveLetter Property
DriveType Property
FileSystem Property
FreeSpace Property
IsReady Property
Path Property
RootFolder Property
SerialNumber Property
ShareName Property
TotalSize Property
VolumeName Property
Microsoft® JScript® 

Drive Object

Methods

The Drive object has no methods.
See Also

- Attributes Property
- DateCreated Property
- DateLastAccessed Property
- DateLastModified Property
- Files Property
- IsRootFolder Property
- Name Property
- ParentFolder Property
- Path Property
- ShortName Property
- ShortPath Property
- Size Property
- SubFolders Property
- Type Property
Microsoft® JScript® Drive Property

Applies To

File Object
Folder Object
DriveExists

Method

See Also

Drive Object
Drives Collection
FileExists Method
FolderExists Method
GetDrive Method
GetDriveName Method
IsReady Property
Microsoft® JScript® DriveExists Method
Applies To FileSystemObject Object
DriveLetter Property

See Also

- AvailableSpace Property
- DriveType Property
- FileSystem Property
- FreeSpace Property
- IsReady Property
- Path Property
- RootFolder Property
- SerialNumber Property
- ShareName Property
- TotalSize Property
- VolumeName Property
DriveLetter

Property

Applies To

Drive Object
Microsoft® JScript® Drives Collection

See Also

Drive Object
Drives Property
File Object
Files Collection
Folder Object
Folders Collection
Microsoft® JScript® Drives Collection Properties

Count Property
Item Property
Microsoft® JScript® Drives Collection

Methods

The Drives collection has no methods.
See Also

- Drives Collection
- Files Property
- SubFolders Property
Microsoft® JScript® Drives Property

Applies To

FileSystemObject Object
DriveType Property

See Also

AvailableSpace Property
DriveLetter Property
FileSystem Property
FreeSpace Property
IsReady Property
Path Property
RootFolder Property
SerialNumber Property
ShareName Property
TotalSize Property
VolumeName Property
Microsoft® JScript®

DriveType

Property
Applies To

Drive Object
Microsoft® JScript® Exists Method

See Also

- Add Method (Dictionary)
- Items Method
- Keys Method
- Remove Method
- RemoveAll Method
Microsoft® JScript®

**Exists Method**

Applies To

---

*Dictionary Object*
File Object

See Also

Drive Object
Drives Collection
Files Collection
Folder Object
Folders Collection
File Object

Properties

Attributes Property
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drive Property
Name Property
ParentFolder Property
Path Property
ShortName Property
ShortPath Property
Size Property
Type Property
Microsoft® JScript® File Object Methods

Copy Method
Delete Method
Move Method
OpenAsTextStream Method
FileExists Method

See Also

DriveExists Method
FolderExists Method
GetFile Method
GetFileName Method
Microsoft® JScript® FileExists Method Applies To FileSystemObject Object
Microsoft® JScript®

Files Collection

See Also

Drive Object
Drives Collection
File Object
Folder Object
Folders Collection
Microsoft® JScript® Files Collection Properties

Count Property
Item Property
The **Files** collection has no methods.
The following table lists the version of Microsoft JScript implemented by host applications.

<table>
<thead>
<tr>
<th>Host Application</th>
<th>JScript Version</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>Microsoft Internet Explorer 3.0</td>
<td>x</td>
</tr>
<tr>
<td>Microsoft Internet Information Server 1.0</td>
<td></td>
</tr>
<tr>
<td>Microsoft Internet Explorer 4.0</td>
<td></td>
</tr>
<tr>
<td>Microsoft Internet Information Server 4.0</td>
<td></td>
</tr>
<tr>
<td>Microsoft Windows Scripting Host 1.0</td>
<td></td>
</tr>
<tr>
<td>Microsoft Visual Studio 6.0</td>
<td></td>
</tr>
<tr>
<td>Microsoft Internet Explorer 5.0</td>
<td></td>
</tr>
<tr>
<td>Microsoft Internet Information Services 5.0</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Language Element</th>
<th>Version First Introduced</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>$1...$9 Properties</td>
<td></td>
</tr>
<tr>
<td>abs Method</td>
<td></td>
</tr>
<tr>
<td>acos Method</td>
<td></td>
</tr>
<tr>
<td>ActiveXObject Object</td>
<td></td>
</tr>
<tr>
<td>Addition Operator (+)</td>
<td></td>
</tr>
<tr>
<td>anchor Method</td>
<td></td>
</tr>
<tr>
<td>Argument/Method/Operator</td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td></td>
</tr>
<tr>
<td>arguments Property</td>
<td></td>
</tr>
<tr>
<td>Array Object</td>
<td></td>
</tr>
<tr>
<td>asin Method</td>
<td></td>
</tr>
<tr>
<td>Assignment Operator (=)</td>
<td></td>
</tr>
<tr>
<td>atan Method</td>
<td></td>
</tr>
<tr>
<td>atan2 Method</td>
<td></td>
</tr>
<tr>
<td>atEnd Method</td>
<td></td>
</tr>
<tr>
<td>big Method</td>
<td></td>
</tr>
<tr>
<td>Bitwise AND Operator (&amp;)</td>
<td></td>
</tr>
<tr>
<td>Bitwise Left Shift Operator (&lt;&lt;)</td>
<td></td>
</tr>
<tr>
<td>Bitwise NOT Operator (~)</td>
<td></td>
</tr>
<tr>
<td>Bitwise OR Operator (</td>
<td>)</td>
</tr>
<tr>
<td>Bitwise Right Shift Operator (&gt;&gt;)</td>
<td></td>
</tr>
<tr>
<td>Bitwise XOR Operator (^)</td>
<td></td>
</tr>
<tr>
<td>blink Method</td>
<td></td>
</tr>
<tr>
<td>bold Method</td>
<td></td>
</tr>
<tr>
<td>Boolean Object</td>
<td></td>
</tr>
<tr>
<td>break Statement</td>
<td></td>
</tr>
<tr>
<td>caller Property</td>
<td></td>
</tr>
<tr>
<td>catch Statement</td>
<td></td>
</tr>
<tr>
<td>@cc_on Statement</td>
<td></td>
</tr>
<tr>
<td>ceil Method</td>
<td></td>
</tr>
<tr>
<td>charAt Method</td>
<td></td>
</tr>
<tr>
<td>charCodeAt Method</td>
<td></td>
</tr>
<tr>
<td>Comma Operator (,)</td>
<td></td>
</tr>
<tr>
<td>// (Single-line Comment Statement)</td>
<td></td>
</tr>
<tr>
<td>/<em>..</em>/ (Multiline Comment)</td>
<td></td>
</tr>
<tr>
<td><strong>Statement)</strong></td>
<td><strong>Comparison Operators</strong></td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td></td>
<td><strong>compile Method</strong></td>
</tr>
<tr>
<td></td>
<td><strong>concat Method (Array)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>concat Method (String)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Conditional Compilation</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Conditional Compilation Variables</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Conditional (trinary) Operator (?:)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>constructor Property</strong></td>
</tr>
<tr>
<td></td>
<td><strong>continue Statement</strong></td>
</tr>
<tr>
<td></td>
<td><strong>cos Method</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Data Type Conversion</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Date Object</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Decrement Operator (--)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>delete Operator</strong></td>
</tr>
<tr>
<td></td>
<td><strong>description Property</strong></td>
</tr>
<tr>
<td></td>
<td><strong>dimensions Method</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Division Operator (/)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>do...while Statement</strong></td>
</tr>
<tr>
<td></td>
<td><strong>E Property</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Enumerator Object</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Equality Operator (==)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Error Object</strong></td>
</tr>
<tr>
<td></td>
<td><strong>escape Method</strong></td>
</tr>
<tr>
<td></td>
<td><strong>eval Method</strong></td>
</tr>
<tr>
<td></td>
<td><strong>exec Method</strong></td>
</tr>
<tr>
<td></td>
<td><strong>exp Method</strong></td>
</tr>
<tr>
<td></td>
<td><strong>fixed Method</strong></td>
</tr>
<tr>
<td>Method</td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---</td>
</tr>
<tr>
<td>floor Method</td>
<td>x</td>
</tr>
<tr>
<td>fontcolor Method</td>
<td>x</td>
</tr>
<tr>
<td>fontsize Method</td>
<td>x</td>
</tr>
<tr>
<td>for Statement</td>
<td>x</td>
</tr>
<tr>
<td>for...in Statement</td>
<td>x</td>
</tr>
<tr>
<td>fromCharCode Method</td>
<td>x</td>
</tr>
<tr>
<td>Function Object</td>
<td>x</td>
</tr>
<tr>
<td>function Statement</td>
<td>x</td>
</tr>
<tr>
<td>getDate Method</td>
<td>x</td>
</tr>
<tr>
<td>getDay Method</td>
<td>x</td>
</tr>
<tr>
<td>getFullYear Method</td>
<td>x</td>
</tr>
<tr>
<td>getHours Method</td>
<td>x</td>
</tr>
<tr>
<td>getItem Method</td>
<td>x</td>
</tr>
<tr>
<td>getMilliseconds Method</td>
<td>x</td>
</tr>
<tr>
<td>getMinutes Method</td>
<td>x</td>
</tr>
<tr>
<td>getMonth Method</td>
<td>x</td>
</tr>
<tr>
<td>GetObject Function</td>
<td>x</td>
</tr>
<tr>
<td>getSeconds Method</td>
<td>x</td>
</tr>
<tr>
<td>getTime Method</td>
<td>x</td>
</tr>
<tr>
<td>getTimezoneOffset Method</td>
<td>x</td>
</tr>
<tr>
<td>getUTCDate Method</td>
<td>x</td>
</tr>
<tr>
<td>getUTCDay Method</td>
<td>x</td>
</tr>
<tr>
<td>getUTCFullYear Method</td>
<td>x</td>
</tr>
<tr>
<td>getUTCHours Method</td>
<td>x</td>
</tr>
<tr>
<td>getUTCMilliseconds Method</td>
<td>x</td>
</tr>
<tr>
<td>getUTCMinutes Method</td>
<td>x</td>
</tr>
<tr>
<td>getUTCMonth Method</td>
<td>x</td>
</tr>
<tr>
<td>getUTCSeconds Method</td>
<td>x</td>
</tr>
<tr>
<td>getVarDate Method</td>
<td>x</td>
</tr>
<tr>
<td>Term</td>
<td></td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>---</td>
</tr>
<tr>
<td><code>getYear Method</code></td>
<td>x</td>
</tr>
<tr>
<td>Global Object</td>
<td>x</td>
</tr>
<tr>
<td>Greater than Operator (<code>&gt;</code> )</td>
<td>x</td>
</tr>
<tr>
<td>Greater than or equal to Operator (<code>&gt;=</code>)</td>
<td>x</td>
</tr>
<tr>
<td>Identity Operator (<code>===</code>)</td>
<td>x</td>
</tr>
<tr>
<td><code>@if</code> Statement</td>
<td>x</td>
</tr>
<tr>
<td>if...else Statement</td>
<td>x</td>
</tr>
<tr>
<td>Increment Operator (<code>++</code>)</td>
<td>x</td>
</tr>
<tr>
<td><code>index</code> Property</td>
<td>x</td>
</tr>
<tr>
<td><code>indexOf</code> Method</td>
<td>x</td>
</tr>
<tr>
<td>Inequality Operator (<code>!==</code>)</td>
<td>x</td>
</tr>
<tr>
<td>Infinity Property</td>
<td>x</td>
</tr>
<tr>
<td><code>input</code> Property</td>
<td>x</td>
</tr>
<tr>
<td><code>instanceof</code> Operator</td>
<td>x</td>
</tr>
<tr>
<td><code>isFinite</code> Method</td>
<td>x</td>
</tr>
<tr>
<td><code>isNaN</code> Method</td>
<td>x</td>
</tr>
<tr>
<td><code>italics</code> Method</td>
<td>x</td>
</tr>
<tr>
<td><code>item</code> Method</td>
<td>x</td>
</tr>
<tr>
<td><code>join</code> Method</td>
<td>x</td>
</tr>
<tr>
<td>Labeled Statement</td>
<td>x</td>
</tr>
<tr>
<td><code>lastIndex</code> Property</td>
<td>x</td>
</tr>
<tr>
<td><code>lastIndexOf</code> Method</td>
<td>x</td>
</tr>
<tr>
<td><code>lbound</code> Method</td>
<td>x</td>
</tr>
<tr>
<td><code>length</code> Property (Array)</td>
<td>x</td>
</tr>
<tr>
<td><code>length</code> Property (Function)</td>
<td>x</td>
</tr>
<tr>
<td><code>length</code> Property (String)</td>
<td>x</td>
</tr>
<tr>
<td>Less than Operator (<code>&lt;</code>)</td>
<td>x</td>
</tr>
<tr>
<td>Less than or equal to Operator (<code>&lt;=</code>)</td>
<td>x</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>link</td>
<td>Method</td>
</tr>
<tr>
<td>LN2</td>
<td>Property</td>
</tr>
<tr>
<td>LN10</td>
<td>Property</td>
</tr>
<tr>
<td>log</td>
<td>Method</td>
</tr>
<tr>
<td>LOG2E</td>
<td>Property</td>
</tr>
<tr>
<td>LOG10E</td>
<td>Property</td>
</tr>
<tr>
<td>Logical AND Operator (&amp;&amp;)</td>
<td></td>
</tr>
<tr>
<td>Logical NOT Operator (!)</td>
<td></td>
</tr>
<tr>
<td>Logical OR Operator (</td>
<td></td>
</tr>
<tr>
<td>match</td>
<td>Method</td>
</tr>
<tr>
<td>Math Object</td>
<td></td>
</tr>
<tr>
<td>max</td>
<td>Method</td>
</tr>
<tr>
<td>MAX_VALUE</td>
<td>Property</td>
</tr>
<tr>
<td>min</td>
<td>Method</td>
</tr>
<tr>
<td>MIN_VALUE</td>
<td>Property</td>
</tr>
<tr>
<td>Modulus Operator (%)</td>
<td></td>
</tr>
<tr>
<td>moveFirst</td>
<td>Method</td>
</tr>
<tr>
<td>moveNext</td>
<td>Method</td>
</tr>
<tr>
<td>Multiplication Operator (*)</td>
<td></td>
</tr>
<tr>
<td>NaN Property (Global)</td>
<td></td>
</tr>
<tr>
<td>NaN Property (Number)</td>
<td></td>
</tr>
<tr>
<td>NEGATIVE_INFINITY Property</td>
<td></td>
</tr>
<tr>
<td>new Operator</td>
<td></td>
</tr>
<tr>
<td>Nonidentity Operator (!==)</td>
<td></td>
</tr>
<tr>
<td>Number Object</td>
<td></td>
</tr>
<tr>
<td>number Property</td>
<td></td>
</tr>
<tr>
<td>Object Object</td>
<td></td>
</tr>
<tr>
<td>Operator Precedence</td>
<td></td>
</tr>
<tr>
<td>Term</td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>--</td>
</tr>
<tr>
<td>parse Method</td>
<td>x</td>
</tr>
<tr>
<td>parseFloat Method</td>
<td>x</td>
</tr>
<tr>
<td>parseInt Method</td>
<td>x</td>
</tr>
<tr>
<td>PI Property</td>
<td>x</td>
</tr>
<tr>
<td>POSITIVE_INFINITY</td>
<td>x</td>
</tr>
<tr>
<td>Property</td>
<td></td>
</tr>
<tr>
<td>pow Method</td>
<td>x</td>
</tr>
<tr>
<td>prototype Property</td>
<td>x</td>
</tr>
<tr>
<td>random Method</td>
<td>x</td>
</tr>
<tr>
<td>RegExp Object</td>
<td></td>
</tr>
<tr>
<td>Regular Expression Object</td>
<td></td>
</tr>
<tr>
<td>Regular Expression Syntax</td>
<td>x</td>
</tr>
<tr>
<td>replace Method</td>
<td>x</td>
</tr>
<tr>
<td>return Statement</td>
<td>x</td>
</tr>
<tr>
<td>reverse Method</td>
<td>x</td>
</tr>
<tr>
<td>round Method</td>
<td>x</td>
</tr>
<tr>
<td>ScriptEngine Function</td>
<td>x</td>
</tr>
<tr>
<td>ScriptEngineBuildVersion Function</td>
<td>x</td>
</tr>
<tr>
<td>ScriptEngineMajorVersion Function</td>
<td>x</td>
</tr>
<tr>
<td>ScriptEngineMinorVersion Function</td>
<td>x</td>
</tr>
<tr>
<td>search Method</td>
<td>x</td>
</tr>
<tr>
<td>@set Statement</td>
<td>x</td>
</tr>
<tr>
<td>setDate Method</td>
<td>x</td>
</tr>
<tr>
<td>setFullYear Method</td>
<td>x</td>
</tr>
<tr>
<td>setHours Method</td>
<td>x</td>
</tr>
<tr>
<td>setMilliseconds Method</td>
<td>x</td>
</tr>
<tr>
<td>setMinutes Method</td>
<td>x</td>
</tr>
<tr>
<td>Method/Property</td>
<td></td>
</tr>
<tr>
<td>---------------------------------</td>
<td>---</td>
</tr>
<tr>
<td>setMonth Method</td>
<td>x</td>
</tr>
<tr>
<td>setSeconds Method</td>
<td>x</td>
</tr>
<tr>
<td>setTime Method</td>
<td>x</td>
</tr>
<tr>
<td>setUTCDate Method</td>
<td></td>
</tr>
<tr>
<td>setUTCFullYear Method</td>
<td></td>
</tr>
<tr>
<td>setUTCHours Method</td>
<td></td>
</tr>
<tr>
<td>setUTCMilliseconds Method</td>
<td></td>
</tr>
<tr>
<td>setUTCMilliseconds Method</td>
<td></td>
</tr>
<tr>
<td>setUCMonth Method</td>
<td></td>
</tr>
<tr>
<td>setUTCSeconds Method</td>
<td></td>
</tr>
<tr>
<td>setYear Method</td>
<td>x</td>
</tr>
<tr>
<td>sin Method</td>
<td>x</td>
</tr>
<tr>
<td>slice Method (Array)</td>
<td>x</td>
</tr>
<tr>
<td>slice Method (String)</td>
<td>x</td>
</tr>
<tr>
<td>small Method</td>
<td>x</td>
</tr>
<tr>
<td>sort Method</td>
<td></td>
</tr>
<tr>
<td>source Property</td>
<td></td>
</tr>
<tr>
<td>split Method</td>
<td>x</td>
</tr>
<tr>
<td>sqrt Method</td>
<td>x</td>
</tr>
<tr>
<td>SQRT1_2 Property</td>
<td>x</td>
</tr>
<tr>
<td>SQRT2 Property</td>
<td>x</td>
</tr>
<tr>
<td>strike Method</td>
<td>x</td>
</tr>
<tr>
<td>String Object</td>
<td>x</td>
</tr>
<tr>
<td>sub Method</td>
<td>x</td>
</tr>
<tr>
<td>substr Method</td>
<td>x</td>
</tr>
<tr>
<td>substring Method</td>
<td>x</td>
</tr>
<tr>
<td>Subtraction Operator (-)</td>
<td>x</td>
</tr>
<tr>
<td>sup Method</td>
<td>x</td>
</tr>
<tr>
<td>switch Statement</td>
<td>x</td>
</tr>
<tr>
<td>tan Method</td>
<td>x</td>
</tr>
<tr>
<td>Keyword</td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>---</td>
</tr>
<tr>
<td>test Method</td>
<td></td>
</tr>
<tr>
<td>this Statement</td>
<td>x</td>
</tr>
<tr>
<td>throw Statement</td>
<td></td>
</tr>
<tr>
<td>toArray Method</td>
<td>x</td>
</tr>
<tr>
<td>toGMTString Method</td>
<td>x</td>
</tr>
<tr>
<td>toLocaleString Method</td>
<td>x</td>
</tr>
<tr>
<td>toLowerCase Method</td>
<td>x</td>
</tr>
<tr>
<td>toString Method</td>
<td>x</td>
</tr>
<tr>
<td>toUpperCase Method</td>
<td>x</td>
</tr>
<tr>
<td>toUTCString Method</td>
<td>x</td>
</tr>
<tr>
<td>try Statement</td>
<td></td>
</tr>
<tr>
<td>typeof Operator</td>
<td>x</td>
</tr>
<tr>
<td>ubound Method</td>
<td></td>
</tr>
<tr>
<td>Unary Negation Operator (−)</td>
<td>x</td>
</tr>
<tr>
<td>unescape Method</td>
<td>x</td>
</tr>
<tr>
<td>Unsigned Right Shift Operator (&gt;&gt;&gt;)</td>
<td>x</td>
</tr>
<tr>
<td>UTC Method</td>
<td>x</td>
</tr>
<tr>
<td>valueOf Method</td>
<td>x</td>
</tr>
<tr>
<td>var Statement</td>
<td>x</td>
</tr>
<tr>
<td>VBArray Object</td>
<td>x</td>
</tr>
<tr>
<td>void Operator</td>
<td>x</td>
</tr>
<tr>
<td>while Statement</td>
<td>x</td>
</tr>
<tr>
<td>with Statement</td>
<td>x</td>
</tr>
</tbody>
</table>
See Also

Attributes Property
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drive Property
IsRootFolder Property
Name Property
ParentFolder Property
Path Property
ShortName Property
ShortPath Property
Size Property
SubFolders Property
Type Property
Microsoft® JScript® Files Property

Applies To

Folder Object
Microsoft® JScript® FileSystemObject

Object

Methods

BuildPath Method
CopyFile Method
CopyFolder Method
CreateFolder Method
CreateTextFile Method
DeleteFile Method
DeleteFolder Method
DriveExists Method
FileExists Method
FolderExists Method
GetAbsolutePathName Method
GetBaseName Method
GetDrive Method
GetDriveName Method
GetExtensionName Method
GetFile Method
GetFileName Method
GetFolder Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
MoveFile Method
MoveFolder Method
OpenTextFile Method
Microsoft® JScript® FileSystemObject

Object

Properties

Drives Property
See Also

AvailableSpace Property
DriveLetter Property
DriveType Property
FreeSpace Property
IsReady Property
Path Property
RootFolder Property
SerialNumber Property
ShareName Property
TotalSize Property
VolumeName Property
Microsoft® JScript® FileSystem

Property Applies To

Drive Object
Folder Object

See Also

Drive Object
Drives Collection
File Object
Files Collection
Folders Collection
Folder Object

Properties

Attributes Property
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drive Property
Files Property
IsRootFolder Property
Name Property
ParentFolder Property
Path Property
ShortName Property
ShortPath Property
Size Property
SubFolders Property
Type Property
Folder Object

Methods

Copy Method
Delete Method
Move Method
OpenAsTextStream Method
Microsoft® JScript®

Folders Collection

See Also

Drive Object
Drives Collection
File Object
Files Collection
Folder Object
SubFolders Property
## Folders Collection Properties

<table>
<thead>
<tr>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Count Property</strong></td>
</tr>
<tr>
<td><strong>Item Property</strong></td>
</tr>
</tbody>
</table>

[Scripting Run-Time Reference](#)
Add Method (Folders)
Microsoft® JScript®

FolderExists Method

See Also

DriveExists Method
FileExists Method
GetFolder Method
GetParentFolderName Method
Microsoft® JScript®

**FolderExists**

**Method**

**Applies To**

**FileSystemObject Object**
FreeSpace Property

See Also

AvailableSpace Property
DriveLetter Property
DriveType Property
FileSystem Property
IsReady Property
Path Property
RootFolder Property
SerialNumber Property
ShareName Property
TotalSize Property
VolumeName Property
Microsoft® JScript® FreeSpace Property Applies To Drive Object
GetAbsolutePathName Method

See Also

GetBaseName Method
GetDrive Method
GetDriveName Method
GetExtensionName Method
GetFile Method
GetFileName Method
GetFileVersion Method
GetFolder Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
GetAbsolutePathName Method
Applies To

FileSystemObject Object
GetBaseName Method

See Also

GetAbsolutePathName Method
GetDrive Method
GetDriveName Method
GetExtensionName Method
GetFile Method
GetFileName Method
GetFileVersion Method
GetFolder Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
Microsoft® JScript® GetBaseName

Method
Applies To

FileSystemObject Object
Microsoft® JScript®

GetDrive Method

See Also

GetAbsolutePathName Method
GetBaseName Method
GetDriveName Method
GetExtensionName Method
GetFile Method
GetFileName Method
GetFileVersion Method
GetFolder Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
Microsoft® JScript® GetDrive Method

Applies To

FileSystemObject Object
GetDriveName Method

See Also

GetAbsolutePathName Method
GetBaseName Method
GetDrive Method
GetExtensionName Method
GetFile Method
GetFileName Method
GetFileVersion Method
GetFolder Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
Microsoft® JScript® 

**GetDriveName**

Method

Applies To

**FileSystemObject Object**
GetExtensionName Method

See Also

GetAbsolutePathName Method
GetBaseName Method
GetDrive Method
GetDriveName Method
GetFile Method
GetFileName Method
GetFileVersion Method
GetFolder Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
GetExtensionName Method
Applies To

FileSystemObject Object
GetFile Method

See Also

GetAbsolutePathName Method
GetBaseName Method
GetDrive Method
GetDriveName Method
GetExtensionName Method
GetFileName Method
GetFileVersion Method
GetFolder Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
Microsoft® JScript® GetFile Method

Applies To

FileSystemObject Object
GetFileName Method

See Also

GetAbsolutePathName Method
GetBaseName Method
GetDrive Method
GetDriveName Method
GetExtensionName Method
GetFile Method
GetFileVersion Method
GetFolder Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
Microsoft® JScript® GetFileName Method
Applies To

FileSystemObject Object
GetFileVersion Method

See Also

GetAbsolutePathName Method
GetBaseName Method
GetDrive Method
GetDriveName Method
GetExtensionName Method
GetFile Method
GetFileName Method
GetFolder Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
Microsoft® JScript® GetFileVersion
Method
Applies To

FileSystemObject Object
Microsoft® JScript®

GetFolder Method

See Also

GetAbsolutePathName Method
GetBaseName Method
GetDrive Method
GetDriveName Method
GetExtensionName Method
GetFile Method
GetFileName Method
GetFileVersion Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
Microsoft® JScript® GetFolder Method Applies To FileSystemObject Object
GetParentFolderName Method

See Also

GetAbsolutePathName Method
GetBaseName Method
GetDrive Method
GetDriveName Method
GetExtensionName Method
GetFile Method
GetFileName Method
GetFileVersion Method
GetFolder Method
GetSpecialFolder Method
GetTempName Method
GetParentFolderName Method
Applies To

FileSystemObject Object
Microsoft® JScript®

GetSpecialFolder

Method

See Also

GetAbsolutePathName Method
GetBaseName Method
GetDrive Method
GetDriveName Method
GetExtensionName Method
GetFile Method
GetFileName Method
GetFileVersion Method
GetFolder Method
GetParentFolderName Method
GetTempName Method

Scripting Run-Time Reference
Microsoft® JScript® GetSpecialFolder Method Applies To FileSystemObject Object

Scripting Run-Time Reference
GetTempName Method

See Also

GetAbsolutePathName Method
GetBaseName Method
GetDrive Method
GetDriveName Method
GetExtensionName Method
GetFile Method
GetFileName Method
GetFileVersion Method
GetFolder Method
GetParentDirectoryName Method
GetSpecialFolder Method
Microsoft® JScript® GetTempName Method Applies To FileSystemObject Object
Microsoft® JScript® IsReady Property

See Also

AvailableSpace Property
DriveLetter Property
DriveType Property
FileSystem Property
FreeSpace Property
Path Property
RootFolder Property
SerialNumber Property
ShareName Property
TotalSize Property
VolumeName Property
<table>
<thead>
<tr>
<th>Microsoft® JScript®</th>
<th><strong>IsReady Property</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Applies To</strong></td>
<td><strong>Drive Object</strong></td>
</tr>
</tbody>
</table>
IsRootFolder Property

See Also

Attributes Property
createdAt Property
DateLastAccessed Property
DateLastModified Property
Drive Property
Files Property
Name Property
ParentFolder Property
Path Property
ShortName Property
ShortPath Property
Size Property
SubFolders Property
Type Property
Microsoft® JScript® IsRootFolder Property Applies To Folder Object
Microsoft® JScript® Items Method

See Also

Add Method (Dictionary)
Exists Method
Keys Method
Remove Method
RemoveAll Method
Items Method

Appplies To

Dictionary Object
Microsoft® JScript®

Keys Method

See Also

- **Add Method (Dictionary)**
- **Exists Method**
- **Items Method**
- **Remove Method**
- **RemoveAll Method**
Microsoft® JScript® *Keys Method*

Applies To

**Dictionary Object**
Microsoft® JScript®  

**Line Property**

**See Also**

- Column Property
Microsoft® JScript® Line Property

Applies To

TextStream Object
Microsoft® JScript® Move Method

See Also

Copy Method
Delete Method
MoveFile Method
MoveFolder Method
OpenAsTextStream Method
Microsoft® JScript® Move Method

Applies To

File Object
Folder Object
Microsoft® JScript®  MoveFile Method

See Also

CopyFile Method
DeleteFile Method
GetFile Method
GetFileName Method
Move Method
MoveFolder Method
OpenTextFile Method

Scripting Run-Time Reference
Microsoft® JScript® MoveFile Method

Applies To

FileSystemObject Object
Microsoft® JScript® MoveFolder Method

See Also

- CopyFile Method
- DeleteFile Method
- GetFile Method
- GetFileName Method
- Move Method
- MoveFile Method
- OpenTextFile Method
Microsoft® JScript® MoveFolder Method Applies To FileSystemObject Object
See Also

Attributes Property
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drive Property
Files Property
IsRootFolder Property
ParentFolder Property
Path Property
ShortName Property
ShortPath Property
Size Property
SubFolders Property
Type Property
Microsoft® JScript® Name Property

Appplies To

File Object
Folder Object
Microsoft® JScript® OpenTextFile Method

See Also

CreateTextFile Method
OpenAsTextStream Method

Scripting Run-Time Reference
Microsoft® JScript® OpenTextFile Method Applies To FileSystemObject Object
ParentFolder Property

See Also

Attributes Property
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drive Property
Files Property
IsRootFolder Property
Name Property
Path Property
ShortName Property
ShortPath Property
Size Property
SubFolders Property
Type Property
Microsoft® JScript® ParentFolder Property Applies To

File Object
Folder Object
See Also

Attributes Property
AvailableSpace Property
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drive Property
DriveLetter Property
DriveType Property
Files Property
FileSystem Property
FreeSpace Property
IsReady Property
IsRootFolder Property
Name Property
ParentFolder Property
RootFolder Property
SerialNumber Property
ShareName Property
ShortName Property
ShortPath Property
Size Property
SubFolders Property
TotalSize Property
Type Property
VolumeName Property
Microsoft® JScript® Path Property

Applies To

Drive Object
File Object
Folder Object
Microsoft® JScript®

Read Method

See Also

ReadAll Method
ReadLine Method
Skip Method
SkipLine Method
Microsoft® JScript® Read Method

Applies To

TextStream Object
Microsoft® JScript®

ReadLine Method

See Also

Read Method
ReadAll Method
Skip Method
SkipLine Method
Microsoft® JScript® `ReadLine`

**Method**

Applies To

**TextStream Object**
**Remove Method**

See Also

- Add Method (Dictionary)
- Exists Method
- Items Method
- Keys Method
- RemoveAll Method
Microsoft® JScript® Remove Method

Applies To

Dictionary Object
RootFolder Property

See Also

AvailableSpace Property
DriveLetter Property
DriveType Property
FileSystem Property
FreeSpace Property
IsReady Property
Path Property
SerialNumber Property
ShareName Property
TotalSize Property
VolumeName Property
Microsoft® JScript® SerialNumber

Property
See Also

AvailableSpace Property
DriveLetter Property
DriveType Property
FileSystem Property
FreeSpace Property
IsReady Property
Path Property
RootFolder Property
ShareName Property
TotalSize Property
VolumeName Property
### Microsoft® JScript® SerialNumber

**Property**

**Applies To**

- **Drive Object**
Property

See Also

AvailableSpace Property
DriveLetter Property
DriveType Property
FileSystem Property
FreeSpace Property
IsReady Property
Path Property
RootFolder Property
SerialNumber Property
TotalSize Property
VolumeName Property
<table>
<thead>
<tr>
<th>Property</th>
<th>Applies To</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShareName</td>
<td>Drive Object</td>
</tr>
</tbody>
</table>
ShortName

Property

See Also

Attributes Property
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drive Property
Files Property
IsRootFolder Property
Name Property
ParentFolder Property
Path Property
ShortPath Property
Size Property
SubFolders Property
Type Property
Microsoft® JScript® ShortName

Property

Applies To

File Object
Folder Object
Microsoft® JScript®

ShortPath

Property

See Also

Attributes Property
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drive Property
Files Property
IsRootFolder Property
Name Property
ParentFolder Property
Path Property
ShortName Property
Size Property
SubFolders Property
Type Property
Microsoft® JScript® ShortPath Property Applies To

File Object
Folder Object
Microsoft® JScript®

**Size Property**

See Also

- Attributes Property
- DateCreated Property
- DateLastAccessed Property
- DateLastModified Property
- Drive Property
- Files Property
- IsRootFolder Property
- Name Property
- ParentFolder Property
- Path Property
- ShortName Property
- ShortPath Property
- SubFolders Property
- Type Property
Microsoft® JScript®  Size Property

Applies To

- File Object
- Folder Object
**Skip Method**

**See Also**

- Close Method
- Read Method
- ReadAll Method
- ReadLine Method
- SkipLine Method
- Write Method
- WriteLine Method
- WriteBlankLines Method
Microsoft® JScript® Skip Method

Applies To

TextStream Object
SubFolders Property
See Also

Attributes Property
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drive Property
Files Property
IsRootFolder Property
Name Property
ParentFolder Property
Path Property
ShortName Property
ShortPath Property
Size Property
Type Property
Microsoft® JScript® SubFolders Property Applies To Folder Object
Microsoft® JScript® TextStream

Object

Methods

- Close Method
- Read Method
- ReadAll Method
- ReadLine Method
- Skip Method
- SkipLine Method
- Write Method
- WriteBlankLines Method
- WriteLine Method
Microsoft® JScript® TextStream Object

Properties

- AtEndOfLine Property
- AtEndOfStream Property
- Column Property
- Line Property
TotalSize Property

See Also

AvailableSpace Property
DriveLetter Property
DriveType Property
FileSystem Property
FreeSpace Property
IsReady Property
Path Property
RootFolder Property
SerialNumber Property
ShareName Property
VolumeName Property
Microsoft® JScript® TotalSize

Property Applies To

Drive Object
Attributes Property
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drive Property
Files Property
IsRootFolder Property
Name Property
ParentFolder Property
Path Property
ShortName Property
ShortPath Property
Size Property
SubFolders Property
Microsoft® JScript®

Type Property

Applies To

File Object
Folder Object
**VolumeName Property**

**See Also**

AvailableSpace Property
DriveLetter Property
DriveType Property
FileSystem Property
FreeSpace Property
IsReady Property
Path Property
RootFolder Property
SerialNumber Property
ShareName Property
TotalSize Property
Microsoft® JScript®

**VolumeName Property**

Applies To

---

**Drive Object**
Microsoft® JScript® Write Method

See Also

WriteBlankLines Method
WriteLine Method
Microsoft® JScript® Write Method
Applies To

TextStream Object
Microsoft® JScript® **WriteBlankLines**

**Method**

**See Also**

- Write Method
- WriteLine Method
Microsoft® JScript® WriteBlankLines Method Applies To TextStream Object
Microsoft® JScript® WriteLine Method

See Also

Write Method
WriteBlankLines Method

Scripting Run-Time Reference
Microsoft® JScript® **WriteLine**

**Method**

Applies To

**TextStream Object**
Microsoft® JScript® getDate Method

Applies To

Date Object
Microsoft® JScript® getDay Method

Applies To

Date Object
Microsoft® JScript®

getHours Method

See Also

Date Object Methods
getUTCHours Method
setHours Method
setUTCHours Method

Language Reference
Microsoft® JScript®

getHours Method

Applies To

Date Object
getMinutes Method

See Also

Date Object Methods
getUTCMinutes Method
setMinutes Method
setUTCMinutes Method
Microsoft® JScript® getMinutes Method Applies To Date Object
getMonth Method

See Also

Date Object Methods
getUTCMonth Method
setMonth Method
setUTCMonth Method
Microsoft® JScript® 

**getMonth**

**Method**

Applies To

**Date Object**
getSeconds Method

See Also

Date Object Methods
getUTCSeconds Method
setSeconds Method
setUTCSeconds Method
Microsoft® JScript® getSeconds
Method
Applies To

Date Object

Language Reference
getTime Method

See Also

Date Object Methods
setTime Method
Microsoft® JScript® getTime Method

Applies To

Date Object

Language Reference
getTimezoneOffset Method
Applies To

Date Object
getYear Method

See Also

Date Object Methods
getFullYear Method
getUTCFullYear Method
setFullYear Method
setUTCFullYear Method
setYear Method
Microsoft® JScript® getYear Method

Applies To

Date Object

Language Reference
Microsoft® JScript®

getUTCFullYear

Method

See Also

Date Object Methods
getFullYear Method
setFullYear Method
setUTCFullYear Method
Microsoft® JScript® **getUTCFullYear**

**Method**

**Applies To**

*Date Object*

[Language Reference](#)
Microsoft® JScript®

getUTCHours Method

See Also

Date Object Methods
getHours Method
setHours Method
setUTCHours Method
Microsoft® JScript® getUTCHours

Method
Applies To

Date Object

Language Reference
Microsoft® JScript®

getUTCMinutes Method

See Also

Date Object Methods
getMinutes Method
setMinutes Method
setUTCMinutes Method

Language Reference
Microsoft® JScript® **getUTCMinutes**

Method

Applies To

---

**Date Object**

---
Microsoft® JScript®

**getUTCMonth**

**Method**

See Also

- [Date Object Methods](#)
- [getMonth Method](#)
- [setMonth Method](#)
- [setUTCMonth Method](#)
Microsoft® JScript® getUTCMonth Method Applies To Language Reference

Date Object
Microsoft® JScript® `getUTCSeconds`

**Method**

**See Also**

- Date Object Methods
- `getSeconds Method`
- `setSeconds Method`
- `setUTCSeconds Method`

[Language Reference](#)
**Microsoft® JScript®**  
**getUTCSeconds**  

**Method**  
**Applies To**  

| Date Object | Language Reference |
Microsoft® JScript® `setDate` Method

Applies To

Date Object
Microsoft® JScript® **setHours Method**

**Applies To**

**Date Object**

[Language Reference]
Microsoft® JScript® setMonth Method

Applies To

Date Object
Microsoft® JScript® *setSeconds*

**Method**

Applies To

---

**Date Object**
Microsoft® JScript® **setTime Method**

Applies To

Date Object
**setYear Method**

**See Also**

- **Date Object Methods**
  - getFullYear Method
  - getUTCFullYear Method
  - getYear Method
  - setFullYear Method
  - setUTCFullYear Method
Microsoft® JScript® *setYear Method*

Applies To

**Date Object**
Microsoft® JScript® 

**setUTCFullYear**

**Method**

**See Also**

- Date Object Methods
- getFullYear Method
- getUTCFullYear Method
- setFullYear Method
Microsoft® JScript® **setUTCFullYear**

**Method**

**Applies To**

**Date Object**
Microsoft® JScript®

**setUTCHours Method**

See Also

- [Date Object Methods](#)
- [getHours Method](#)
- [getUTCHours Method](#)
- [setHours Method](#)
Microsoft® JScript® setUTCHours

Method

Applies To

Date Object
Microsoft® JScript® setUTCMinutes Method

See Also

Date Object Methods
getMinutes Method
getUTCMinutes Method
setMinutes Method
Microsoft® JScript® *setUTCMinutes*

**Method**

Applies To

---

**Date Object**

---

[Language Reference]
Microsoft® JScript® *setUTCMonth*

**Method**

**Applies To**

*Date Object*
Microsoft® JScript® setUTCSeconds Method

See Also

Date Object Methods
getSeconds Method
getUTCSeconds Method
setSeconds Method
Microsoft® JScript® setUTCSeconds Method Applies To Date Object
Microsoft® JScript® **toLocaleString**

**Method**

Applies To

**Date Object**
Microsoft® JScript® parse Method

See Also

Date Object Methods
Microsoft® JScript® parse Method

Applies To

Date Object
UTC Method

See Also

Date Object Methods
setTime Method
Microsoft® JScript® UTC Method

Applies To

Date Object
function Statement
See Also

new Operator
Microsoft® JScript® `parseInt` Method

Applies To

Global Object
Microsoft® JScript® parseFloat Method Applies To Global Object
Microsoft® JScript® % Operator

See Also

% = Operator
Operator Behavior
Operator Precedence
Operator Summary
Microsoft® JScript®

Operator

See Also

*=
Operator
Operator Behavior
Operator Precedence
Operator Summary

Language Reference
See Also

/\ Operator
Operator Behavior
Operator Precedence
Operator Summary
Microsoft® JScript® Global Object

See Also

Object Object
Microsoft® JScript® Global Object Methods

- escape Method
- eval Method
- isFinite Method
- isNaN Method
- parseFloat Method
- parseInt Method
- unescape Method
Microsoft® JScript® Global Object Properties

Infinity Property
NaN Property
Microsoft® JScript® *split Method*

See Also

concat Method  
RegExp Object  
Regular Expression Syntax  
String Object Methods

Language Reference
Microsoft® JScript® *split Method*

**Applies To**

[String Object]
getVarDate Method

See Also

getDate Method
parse Method
**getVarDate** Method

Applies To

Date Object
See Also

exec Method
RegExp Object
replace Method
search Method
String Object Methods
test Method
replace Method

See Also

exec Method
match Method
RegExp Object
search Method
String Object Methods
test Method
Microsoft® JScript® replace Method

Applies To

String Object
OpenAsTextStream Method

See Also

Copy Method
CreateTextFile Method
Delete Method
Move Method
OpenTextFile Method
OpenAsTextStream Method
Applies To

File Object
Members of String.prototype

anchor Method
big Method
blink Method
bold Method
charAt Method
charCodeAt Method
concat Method
fixed Method
fontcolor Method
fontsize Method
fromCharCode Method
indexOf Method
italics Method
lastIndexOf Method
link Method
match Method
replace Method
search Method
slice Method
small Method
split Method
strike Method
substr Method
substr Method
substring Method
sup Method
toLowerCase Method
toUpperCase Method
toString Method
valueOf Method

Nonmembers of String.prototype

The **String** object has no methods that are not part of the prototype.
Computational

Addition (+)
Decrement (--)
Division (/)
Increment (++)
Modulus (%)
Multiplication (*)
Subtraction (-)
Unary negation (-)

Logical

Comma (,)
Conditional (trinary) (?:)
Equality (==)
Greater than (>)
Greater than or equal to (>=)
Identity (===)
Inequality (!=)
Less than (<)
Less than or equal to (<=)
Logical AND (&&)
Logical NOT (!)
Logical OR (||)
Nonidentity (!==)

Bitwise
Bitwise AND (&)
Bitwise Left Shift (<<)
Bitwise NOT (~)
Bitwise OR (|)
Bitwise Right Shift (>>)
Bitwise XOR (^)
Unsigned Right Shift (>>>)

Assignment

Assignment (=)
Compound Assignment Operators

Miscellaneous

delete
instanceof
new
typeof
void
Regular Expression Object Methods

- **compile Method**
- **exec Method**
- **test Method**
Regular Expression Object Properties

lastIndex Property
csource Property
See Also

| Operator
Operator Behavior
Operator Precedence
Operator Summary
Microsoft® JScript®

\(^=\) Operator

See Also

^ Operator
Operator Behavior
Operator Precedence
Operator Summary
See Also

/ Operator
Operator Behavior
Operator Precedence
Operator Summary
<<= Operator

See Also

<< Operator
>> Operator
>>> Operator
Operator Behavior
Operator Precedence
Operator Summary

Language Reference
See Also

% Operator
Operator Behavior
Operator Precedence
Operator Summary
See Also

* Operator
Operator Behavior
Operator Precedence
Operator Summary
Microsoft® JScript®

See Also

<< Operator
>> Operator
>>>> Operator

Operator Behavior
Operator Precedence
Operator Summary

Language Reference
See Also

- Operator
Operator Behavior
Operator Precedence
Operator Summary
Microsoft® JScript®

See Also

>>> Operator
<< Operator
>> Operator
Operator Behavior
Operator Precedence
Operator Summary
Description

Returns the special value NaN indicating that an expression is not a number.

Syntax

NaN

Remarks

The NaN property (not a number) is a member of the Global object, and is made available when the scripting engine is initialized.
**RemoveAll** Method

**See Also**  
**Applies To**

---

**Description**

The **RemoveAll** method removes all key, item pairs from a **Dictionary** object.

**Syntax**

```javascript
object.RemoveAll()
```

The `object` is always the name of a **Dictionary** object.

**Remarks**

The following code illustrates use of the **RemoveAll** method:

```javascript
var a, d, i; // Create some variables.
d = new ActiveXObject("Scripting.Dictionary");
d.Add("a", "Athens"); // Add some keys and items.
d.Add("b", "Belgrade");
d.Add("c", "Cairo");
...
d.RemoveAll(); // Clear the dictionary.
```
### Item Property

**Description**

Sets or returns an *item* for a specified *key* in a **Dictionary** object. For collections, returns an *item* based on the specified *key*. Read/write.

**Syntax**

```javascript
object.Item(key)[ = newitem]
```

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

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a collection or <strong>Dictionary</strong> object.</td>
</tr>
<tr>
<td>key</td>
<td>Required. <em>Key</em> associated with the <em>item</em> being retrieved or added.</td>
</tr>
<tr>
<td>newitem</td>
<td>Optional. Used for <strong>Dictionary</strong> object only; no application for collections. If provided, <em>newitem</em> is the new value associated with the specified <em>key</em>.</td>
</tr>
</tbody>
</table>

**Remarks**

If *key* is not found when changing an *item*, a new *key* is created with the specified *newitem*. If *key* is not found when attempting to return an existing *item*, a new *key* is created and its corresponding *item* is left empty.
The following example illustrates the use of the `Item` property.

```javascript
function DicTest(keyword) {
    var a, d;
    d = new ActiveXObject("Scripting.Dictionary");
    d.Add("a", "Athens");
    d.Add("b", "Belgrade");
    d.Add("c", "Cairo");
    a = d.Item(keyword);
    return(a);
}
```
Members of Date.prototype

getDate Method
getDay Method
getFullYear Method
getHours Method
getMilliseconds Method
getMinutes Method
getMonth Method
getSeconds Method
g getTime Method
g getTimezoneOffset Method
g getUTCDate Method
g getUTCDay Method
g getUTCFullYear Method
g getUTCHours Method
g getUTCMilliseconds Method
g getUTCMinutes Method
g getUTCMonth Method
g getUTCSeconds Method
g getVarDate Method
g getYear Method
setDate Method
setFullYear Method
setHours Method
setMilliseconds Method
setMinutes Method
setMonth Method
setSeconds Method
setTime Method
setUTCDate Method
setUTCFullYear Method
setUTCHours Method
setUTCMilliseconds Method
setUTCMinutes Method
setUTCMonth Method
setUTCSeconds Method
setYear Method
toGMTString Method
toLocaleString Method
toUTCString Method

toString Method
valueOf Method

Nonmembers of Date.prototype

parse Method
UTC Method
**getFullYear**

**Method**

**See Also**

**Applies To**

**Description**

Returns the year value in the **Date** object using **local time**.

**Syntax**

```
objDate.getFullYear()
```

**Remarks**

To get the year using **Universal Coordinated Time (UTC)**, use the **getUTCFullYear** method.

The **getFullYear** method returns the year as an absolute number. For example, the year 1976 is returned as 1976. This avoids the classic year 2000 problem where dates beginning with January 1, 2000 are confused with those beginning with January 1, 1900.

The following example illustrates the use of the **GetFullYear** method:

```
function DateDemo()
{
    var d, s = "Today's UTC date is: ";
    d = new Date();
```
s += (d.getMonth() + 1) + "/";

s += d.getDate() + "/";

s += d.getFullYear();

return(s);

}
**Method**

**Description**

Sets the year value in the **Date** object using *local time*.

**Syntax**

```javascript
objDate.setFullYear(numYear[, numMonth[, numDate]])
```

The `setFullYear` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>numYear</code></td>
<td>Required. A numeric value equal to the year.</td>
</tr>
<tr>
<td><code>numMonth</code></td>
<td>Optional. A numeric value equal to the month. Must be supplied if <code>numDate</code> is supplied.</td>
</tr>
<tr>
<td><code>numDate</code></td>
<td>Optional. A numeric value equal to the date.</td>
</tr>
</tbody>
</table>

**Remarks**

All `set` methods taking optional arguments use the value returned from corresponding `get` methods, if you do not specify an optional argument. For example, if the `numMonth` argument is optional, but not specified, JScript uses the value returned from the `getMonth` method.

In addition, if the value of an argument is greater than its range or is a negative number, other stored values are modified accordingly.
To set the year using Universal Coordinated Time (UTC), use the `setUTCFullYear` method.

The range of years supported in the date object is approximately 285,616 years from either side of 1970.

The following example illustrates the use of the `setFullYear` method:

```javascript
function SetFullYearDemo(newyear) {
  var d, s;
  d = new Date();
  d.setFullYear(newyear);
  s = "Current setting is ";
  s += d.toLocaleString();
  return(s);
}
```
**setMinutes Method**

**See Also**  
**Applies To**

**Description**

Sets the minutes value in the **Date** object using local time.

**Syntax**

```javascript
objDate.setMinutes(numMinutes[, numSeconds[, numMilli]])
```

The **setMinutes** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>numMinutes</strong></td>
<td>Required. A numeric value equal to the minutes value.</td>
</tr>
<tr>
<td><strong>numSeconds</strong></td>
<td>Optional. A numeric value equal to the seconds value. Must be supplied if the <strong>numMilli</strong> argument is used.</td>
</tr>
<tr>
<td><strong>numMilli</strong></td>
<td>Optional. A numeric value equal to the milliseconds value.</td>
</tr>
</tbody>
</table>

**Remarks**

All **set** methods taking optional arguments use the value returned from corresponding **get** methods, if you do not specify an optional argument. For example, if the **numMonth** argument is
optional, but not specified, JScript uses the value returned from the `getMonth` method.

To set the minutes value using Universal Coordinated Time (UTC), use the `setUTCMinutes` method.

If the value of an argument is greater than its range or is a negative number, other stored values are modified accordingly. For example, if the stored date is "Jan 5, 1996 00:00:00" and `setMinutes(90)` is called, the date is changed to "Jan 5, 1996 01:30:00." Negative numbers have a similar behavior.

The following example illustrates the use of the `setMinutes` method:

```javascript
function SetMinutesDemo(nmin, nsec)
{
    var d, s;
    var sep = ":";
    d = new Date();
    d.setMinutes(nmin, nsec);
    s = "Current setting is " + d.toLocaleString();
    return(s);
}
```
Description

Specifies the index at which to start the next match.

Syntax

```javascript
rgexp.lastIndex [ = index ]
```

The `lastIndex` property syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rgexp</code></td>
<td>Required. A Regular Expression object. Can be a variable name or a literal.</td>
</tr>
<tr>
<td><code>index</code></td>
<td>The index from which to begin the next search.</td>
</tr>
</tbody>
</table>

Remarks

The `lastIndex` property is modified by the `exec` method, and the `match`, `replace`, and `split` methods of the `String` object.

The following rules apply to values of `lastIndex`:

- If `lastIndex` is greater than the length of the string, the `test` and `exec` methods fail, and `lastIndex` is set to zero.
- If `lastIndex` is equal to the length of the string, the regular expression matches if the pattern matches the empty string. Otherwise, the match fails and `lastIndex` is reset to zero.

- Otherwise, `lastIndex` is set to the next position following the most recent match.
Microsoft® JScript® NaN Property

See Also

isNaN Method
Microsoft® JScript® NaN Property

Applies To

Global Object
RemoveAll Method

See Also

Add Method (Dictionary)
Exists Method
Items Method
Keys Method
Remove Method
Microsoft® JScript®

RemoveAll Method

Applies To

Dictionary Object
**Item Property**

**Applies To**

- Dictionary Object
- Drives Collection
- Files Collection
- Folders Collection