JScript

JScript

JScript User's Guide

Using JScript in Internet Explorer

JScript Language Reference

Build: Topic Version 5.6.9309.1546

JScript

JScript User's Guide

JScript Fundamentals

Advanced JScript

Build: Topic Version 5.6.9309.1546

JScript

JScript Fundamentals

Build: Topic Version 5.6.9309.1546

JScript

What Is JScript?

JScript is the Microsoft implementation of the ECMA 262 language specification (ECMAScript Edition 3). With only a few minor exceptions (to maintain backwards compatibility), JScript is a full implementation of the ECMA standard. This overview is intended to help you get started with JScript.

Using JScript

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

JScript is not a cut-down version of another language (it is only distantly and indirectly related to Java, for example), nor is it a simplification of anything. It is, however, limited. You cannot write stand-alone applications in it, for example, and it has no built-in support for reading or writing files. Moreover, JScript scripts can run only in the presence of an interpreter or "host", such as Active Server Pages (ASP), Internet Explorer, or Windows Script Host.

JScript is a loosely typed language. Loosely typed means you do not have to declare the data types of variables explicitly. In fact, JScript takes it one step further. You cannot explicitly declare data types in JScript. Moreover, in many cases JScript performs conversions automatically when needed. For instance, if you add a number to an item consisting of text (a string), the number is converted to text.

The rest of this user's guide is an overview of JScript features. For full details of the language implementation, consult the language reference.

> **Note** The code in many of the following examples is somewhat more explicit and less dense than code you are likely to find in actual Web pages. The intent here is to clarify the concepts, not to express optimal coding conciseness and style. In any case, there is no shame in writing code that you can read and easily understand six months after you write it.

Build: Topic Version 5.6.9309.1546

JScript

Writing JScript Code

Like many other programming languages, Microsoft JScript is written in text format, and 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 (called "literals"), and expressions.

Statements

A JScript program is a collection of statements. A JScript statement is equivalent to a complete sentence in English. JScript statements combine expressions in such a way that they carry out one complete task.

A statement consists of one or more expressions, keywords, or operators (symbols). Typically, a statement is written on a single line, although a statement can be written over two or more lines. Also, two or more statements can be written on the same line by separating them with semicolons. In general, each new line begins a new statement. It is a good idea to terminate your statements explicitly. You do this with the semicolon (;), which is the JScript statement termination character. Here are two examples of JScript statements.

```
aBird = "Robin"; // Assign the text "Robin" to the variable aBird
var today = new Date(); // Assign today's date to the variable today
```

A group of JScript statements surrounded by braces ({}) is called a block. Statements grouped into a block can generally be treated as a single statement. This means you can use blocks in most places that JScript expects a lone statement. Notable exceptions include the headers of **for** and **while** loops. Notice that the primitive statements within a block end in semicolons, but the block itself does not.

Generally, blocks are used in functions and conditionals. Notice that unlike C++ and some other languages, JScript does not consider a block to be a new scope; only functions create a new scope. In the following example, the first statement begins the definition of a function that consists of a block of five statements. Following the block are three statements that are not surrounded by braces; these statements are not a block, and are therefore not part of the function definition.

```
function convert(inches) {
   feet = inches / 12; // These five statements are in a block.
   miles = feet / 5280;
   nauticalMiles = feet / 6080;
```

```
   cm = inches * 2.54;
   meters = inches / 39.37;
}
km = meters / 1000; // These three statements are not in a block.
kradius = km;
mradius = miles;
```

Comments

A single-line JScript comment begins with a pair of forward slashes (//). Here is an example of a single line comment.

```
aGoodIdea = "Comment your code thoroughly."; // This is a single-line comment.
```

A multiline JScript comment begins with a forward slash and asterisk (/*), and ends with the reverse (*/).

```
/*
This is a multiline comment that explains the preceding code statement.

The statement assigns a value to the aGoodIdea variable. The value,
which is contained between the quote marks, is called a literal. A
literal explicitly and directly contains information; it does not
refer to the information indirectly. The quote marks are not part
of the literal.
*/
```

> **Note** If you attempt to embed one multiline comment within another, JScript interprets the resulting multiline comment in an unexpected way. The */ that marks the end of the embedded multiline comment is interpreted as the end of the whole multiline comment. This means that the text that follows the embedded multiline comment will not be commented out; instead, it will be interpreted as JScript code, and will generate syntax errors.

It is recommended that you write all your comments as blocks of single-line comments. This allows you to comment out large segments of code with a multiline comment later.

```
// This is another multiline comment, written as a series of single-line comments.
// After the statement is executed, you can refer to the content of the aGoodIdea
// variable by using its name, as in the next statement, in which a string literal is
// appended to the aGoodIdea variable by concatenation to create a new variable.

var extendedIdea = aGoodIdea + " You never know when you'll have to figure out what it does.";
```

Assignments and Equality

The equal sign (=) is used in JScript statements to assign values to variables: it is the assignment operator. The left hand operand of the = operator is always an Lvalue. Examples of Lvalues are:

- variables,
- array elements,
- object properties.

The right operand of the = operator is always an Rvalue. Rvalues can be an arbitrary value of any type, including the value of an expression. Here is an example of a JScript assignment statement.

```
anInteger = 3;
```

The JScript compiler interprets this statement as meaning: "Assign the value 3 to the variable **anInteger**," or "**anInteger** takes the value 3."

Be certain you understand the difference between the = operator (assignment) and the == operator (equality). When you want to compare two values to find out if they are equal, use two equals sings (==). This is discussed in detail in Controlling Program Flow.

Expressions

A JScript expression is a 'phrase' of JScript that a JScript interpreter can evaluate to generate a value. The value can be of any valid JScript type - a number, a string, an object, and so on. The simplest expressions are literals. Here are some examples of JScript literal expressions.

```
3.9                      // numeric literal
"Hello!"                 // string literal
false                    // boolean literal
null                     // literal null value
{x:1, y:2}               // Object literal
[1,2,3]                  // Array literal
function(x){return x*x;}  // function literal
```

More complicated expressions can contain variables, function calls, and other expressions. You can combine expressions to create complex expressions using operators. Examples of operators are:

```
+  // additon
-  // subtraction
```

```
*  // multiplication
/  // division
```

Here are some examples of JScript complex expressions.

```
var anExpression = 3 * (4 / 5) + 6;
var aSecondExpression = Math.PI * radius * radius;
var aThirdExpression = aSecondExpression + "%" + anExpression;
var aFourthExpression = "(" + aSecondExpression + ") % (" + anExpression + ")";
```

Build: Topic Version 5.6.9309.1546

JScript

JScript Variables

In any programming language, a piece of data is used to quantify a concept.

```
How old am I?
```

In JScript, a variable is the name you give that concept; it represents the value at a given instant. When you use the variable, you really mean the data it represents. Here is an example:

```
NumberOfDaysLeft = EndDate – TodaysDate;
```

In a mechanical sense, you use variables to store, retrieve, and manipulate all the different values that appear in your scripts. Always create a meaningful variable name; that makes it easy for humans to understand what your scripts do.

Declaring Variables

The first time a variable appears in your script is its declaration. This first mention of the variable sets it up in memory so you can refer to it

later on in your script. Always declare variables before using them. You do this using the **var** keyword.

```
var count;  // a single declaration.
var count, amount, level;  // multiple declarations with a single var keyword.
var count = 0, amount = 100;  // variable declaration and initialization in one statement.
```

If you do not initialize your variable in the **var** statement, it automatically takes on the JScript value **undefined**. Although it is unsafe to do so, it is legal JScript syntax to omit the **var** keyword from your declaration statement. When you do, the JScript interpreter gives the variable global scope visibility. When you declare a variable at the procedure level though, you do not want it to be visible at the global scope; in this case, you must use the **var** keyword in your variable declaration.

Naming Variables

A variable name is an identifier. In JScript, identifiers are used to:

- name variables,
- name functions,
- provide labels for loops.

JScript is a case-sensitive language. This means a variable name such as *myCounter* is different than the variable name *MYCounter*. Variable names can be of any length. The rules for creating legal variable names are as follows:

- The first character must be an ASCII letter (either uppercase or lowercase), or an underscore (_) character. Note that a number cannot be used as the first character.
- Subsequent characters must be letters, numbers, or underscores.
- The variable name must not be a reserved word.

Here are some examples of valid variable names:

```
_pagecount
Part9
Number_Items
```

Here are some examples of invalid variable names:

```
99Balloons // Cannot begin with a number.
Smith&Wesson // The ampersand (&) character is not a valid character for variable names.
```

When you want to declare a variable and initialize it, but do not want to give it any particular value, assign it the JScript value null. Here is an example.

```
var bestAge = null;
var muchTooOld = 3 * bestAge; // muchTooOld has the value 0.
```

If you declare a variable without assigning a value to it, it exists, but has the JScript value undefined. Here is an example.

```
var currentCount;
var finalCount = 1 * currentCount; // finalCount has the value NaN since currentCount is undefined.
```

Note that the main difference between **null** and **undefined** in JScript is that **null** behaves like the number 0, while **undefined** behaves like the special value **NaN** (Not a Number). A **null** value and an **undefined** value will always compare to be equal.

You can declare a variable without using the **var** keyword in the declaration, and assign a value to it. This is an implicit declaration.

```
noStringAtAll = ""; // The variable noStringAtAll is declared implicitly.
```

You cannot use a variable that has never been declared.

```
var volume = length * width; // Error - length and width do not yet exist.
```

Coercion

The JScript interpreter can only evaluate expressions in which the data types of the operands are the same. Without coercion, an expression that attempts to perform an operation on two different data types (a number and a string for example) would produce an erroneous result. But that is not the case with JScript.

JScript is a loosely typed language. This means its variables have no predetermined type (as opposed to strongly typed languages like C++). Instead, JScript variables have a type that corresponds to the type of value they contain. A benefit of this behavior is that it provides you with the flexibility to treat a value as if it were of another type.

In JScript, you can perform operations on values of differing types without fear that the JScript interpreter will raise an exception. Instead, the JScript interpreter automatically changes (coerces) one of the data types to that of the other, then performs the operation. For example:

Operation	Result
Add a number and a string	The number is coerced into a string.

Add a Boolean and a string	The Boolean is coerced into a string.
Add a number and a Boolean	The Boolean is coerced into a number.

Consider the following example.

```
var x = 2000;      // A number.
var y = "Hello";   // A string.
x = x + y;         // the number is coerced into a string.
document.write(x); // Outputs 2000Hello.
```

To explicitly convert a string to an integer, use the parseInt Method. To explicitly convert a string to a number, use the parseFloat Method. Notice that strings are automatically converted to equivalent numbers for comparison purposes, but are left as strings for addition (concatenation).

Build: Topic Version 5.6.9309.1546

JScript

JScript Data Types

In JScript, there are three primary data types, two composite data types, and two special data types.

The primary (primitive) data types are:

- String
- Number
- Boolean

The composite (reference) data types are:

- Object

- Array

The special data types are:

- Null
- Undefined

String Data Type

A string value is a chain of zero or more Unicode characters (letters, digits, and punctuation marks) *strung* together. You use the string data type to represent text in JScript. String literals can be included in your scripts by enclosing them in matching pairs of single or double quotation marks. Double quotation marks can be contained within strings surrounded by single quotation marks, and single quotation marks can be contained within strings surrounded by double quotation marks. The following are examples of strings:

```
"Happy am I; from care I'm free!"
'"Avast, ye lubbers!" roared the technician.'
"42"
'c'
```

Notice that JScript does not have a type to represent a single character. To represent a single character in JScript, you create a string that consists of only one character. A string that contains zero characters ("") is an empty (zero-length) string.

Number Data Type

In JScript, there is no distinction between integer and floating-point values; a JScript number can be either (internally, JScript represent all numbers as floating-point values).

Integer Values

Integer values can be positive whole numbers, negative whole numbers, and 0. They can be represented in base 10 (decimal), base 8 (octal), and base 16 (hexadecimal). Most numbers in JScript are written in decimal. You denote octal integers by prefixing them with a leading "0" (zero). They can contain digits 0 through 7 only. A number with a leading "0", containing the digits "8" and/or "9" is interpreted as a decimal number.

You denote hexadecimal ("hex") integers by prefixing them with a leading "0x" (zero and x|X). They can contain digits 0 through 9, and letters A through F (either uppercase or lowercase) only. The letters A through F are used to represent, as single digits, 10 through 15 in base

10. That is, 0xF is equivalent to 15, and 0x10 is equivalent to 16.

Both octal and hexadecimal numbers can be negative, but cannot have a decimal portion, and cannot be written in scientific (exponential) notation.

Floating-point Values

Floating-point values can be whole numbers with a decimal portion. Additionally, they can be expressed in scientific notation. That is, an uppercase or lowercase "e" is used to represent "ten to the power of". JScript represents numbers using the eight byte IEEE 754 floating-point standard for numerical representation. This means you can write numbers as large as $\pm 1.7976931348623157 \times 10^{308}$, and as small as $\pm 5 \times 10^{-324}$. A number that begins with a single "0" and contains a decimal point is interpreted as a decimal floating-point number.

Notice that a number that begins with "0x" or "00" and contains a decimal point will generate an error. Here are some examples of JScript numbers.

Number	Description	Decimal Equivalent
.0001, 0.0001, 1e-4, 1.0e-4	Four equivalent floating-point numbers.	0.0001
3.45e2	A floating-point number.	345
42	An integer.	42
0378	An integer. Although this looks like an octal number (it begins with a zero), 8 is not a valid octal digit, so the number is treated as a decimal.	378
0377	An octal integer. Notice that although it only appears to be one less than the number above, its actual value is quite different.	255
0.0001	A floating point number. Even though this begins with a zero, it is not an octal number because it has a decimal point.	0.0001
00.0001	This is an error. The two leading zeros mark the number as an octal, but octals are not allowed a decimal component.	N/A (compiler error)
0Xff	A hexadecimal integer.	255
0x37CF	A hexadecimal integer.	14287
0x3e7	A hexadecimal integer. Notice that the 'e' is not treated as exponentiation.	999
0x3.45e2	This is an error. Hexadecimal numbers cannot have decimal parts.	N/A (compiler error)

Additionally, JScript contains numbers with special values. These are:

- NaN (not a number). This is used when a mathematical operation is performed on inappropriate data, such as strings or the undefined value
- Positive Infinity. This is used when a positive number is too large to represent in JScript
- Negative Infinity. This is used when a negative number is too large to represent in JScript
- Positive and Negative 0. JScript differentiates between positive and negative zero.

Boolean Data Type

Whereas the string and number data types can have a virtually unlimited number of different values, the Boolean data type can only have two. They are the literals **true** and **false**. A Boolean value is a truth-value — it expresses the validity of a condition (tells whether the condition is true or not).

Comparisons you make in your scripts always have a Boolean outcome. Consider the following line of JScript code.

```
y = (x == 2000);
```

Here, the value of the variable x is tested to see if it is equal to the number 2000. If it is, the result of the comparison is the Boolean value **true**, which is assigned to the variable y**.** If x is not equal to 2000, then the result of the comparison is the Boolean value **false**.

Boolean values are especially useful in control structures. Here, you combine a comparison that creates a Boolean value directly with a statement that uses it. Consider the following JScript code sample.

```
if (x == 2000)
    z = z + 1;
else
    x = x + 1;
```

The **if/else** statement in JScript performs one action if a Boolean value is **true** (in this case, z = z + 1), and an alternate action if the Boolean value is **false** (x = x + 1).

You can use any expression as a comparative expression. Any expression that evaluates to 0, null, undefined, or an empty string is interpreted as **false**. An expression that evaluates to any other value is interpreted as **true**. For example, you could use an expression such as:

```
if (x = y + z) // This may not do what you expect -- see below!
```

Note that the above line does **not** check if x is equal to y + z, since only a single equal sign (assignment) is used. Instead, the code above assigns the value of y + z to the variable x, and then checks if the result of the entire expression (the value of x) is zero. To check if x is equal

to y + z, use the following code.

```
if (x == y + z) // This is different to the code above!
```

For more information on comparisons, see Controlling Program Flow.

Null Data Type

The **null** data type has only one value in JScript: null. The null keyword cannot be used as the name of a function or variable.

A variable that contains null contains "no value" or "no object." In other words, it holds no valid number, string, Boolean, array, or object. You can erase the contents of a variable (without deleting the variable) by assigning it the null value.

Notice that in JScript, null is not the same as 0 (as it is in C and C++). Also note that the **typeof** operator in JScript will report null values as being of type **Object**, not of type null. This potentially confusing behavior is for backwards compatibility.

Undefined Data Type

The undefined value is returned when you use:

- an object property that does not exist,
- a variable that has been declared, but has never had a value assigned to it.

Notice that you cannot test to see if a variable exists by comparing it to undefined, although you can check if its type is "undefined". In the following code example, assume that the programmer is trying to test if the variable **x** has been declared:

```
// This method will not work
if (x == undefined)
    // do something

// This method also won't work - you must check for
// the string "undefined"
if (typeof(x) == undefined)
    // do something

// This method will work
if (typeof(x) == "undefined")
    // do something
```

Consider comparing the undefined value to null.

```
someObject.prop == null;
```

This comparison is **true**,

- if the property someObject.prop contains the value null,
- if the property someObject.prop does not exist.

To check if an object property exists, you can use the new **in** operator:

```
if ("prop" in someObject)
    // someObject has the property 'prop'
```

Build: Topic Version 5.6.9309.1546

JScript

JScript Operators

JScript has a full range of operators, including arithmetic, logical, bitwise, assignment, as well as some miscellaneous operators.

Computational Operators

Description	Symbol
Unary negation	-
Increment	++
Decrement	—
Multiplication	*
Division	/

Modulus arithmetic	%
Addition	+
Subtraction	-

Logical Operators

Description	Symbol
Logical NOT	!
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=
Equality	==
Inequality	!=
Logical AND	&&
Logical OR	||
Conditional (ternary)	?:
Comma	,
Strict Equality	===
Strict Inequality	!==

Bitwise Operators

Description	Symbol
Bitwise NOT	~
Bitwise Left Shift	<<
Bitwise Right Shift	>>
Unsigned Right Shift	>>>
Bitwise AND	&
Bitwise XOR	^
Bitwise OR	|

Assignment Operators

Description	Symbol
Assignment	=
Compound Assignment	*OP*=

Miscellaneous Operators

Description	Symbol
delete	delete
typeof	typeof
void	void
instanceof	instanceof
new	new
in	in

The difference between == (equality) and === (strict equality) is that the equality operator will coerce values of different types before checking for equality. For example, comparing the string "1" with the number 1 will compare as true. The strict equlity operator, on the other hand, will not coerce values to different types, and so the string "1" will not compare as equal to the number 1.

Primitive strings, numbers, and Booleans are compared by value. If they have the same value, they will compare as equal. Objects (including **Array**, **Function**, **String**, **Number**, **Boolean**, **Error, Date** and **RegExp** objects) compare by reference. Even if two variables of these types have the same value, they will only compare as true if they refer to exactly the same object.

For example:

```
// Two primitive strings with the same value.
var string1 = "Hello";
var string2 = "Hello";

// Two String objects, with the same value.
var StringObject1 = new String(string1);
var StringObject2 = new String(string2);

// This will be true.
if (string1 == string2)
     // do something (this will be executed)

// This will be false.
if (StringObject1 == StringObject2)
```

```
    // do something (this will not be executed)

// To compare the value of String objects,
// use the toString() or valueOf() methods.
if (StringObject1.valueOf() == StringObject2)
     // do something (this will be executed)
```

Build: Topic Version 5.6.9309.1546

JScript

Operator Precedence

Operator precedence is a set of rules in JScript. It controls the order in which operations are performed when an expression is evaluated. Operations with a higher precedence are performed before those with a lower one. For example, multiplication is performed before addition.

The following table lists the JScript operators, ordered from highest to lowest precedence. Operators with the same precedence are evaluated left to right.

Operator	Description
. [] ()	Field access, array indexing, function calls, and expression grouping
++ — - ~ ! delete new typeof void	Unary operators, return data type, object creation, undefined values
* / %	Multiplication, division, modulo division
+ - +	Addition, subtraction, string concatenation
<< >> >>>	Bit shifting
< <= > >= instanceof	Less than, less than or equal, greater than, greater than or equal, instanceof
== != === !==	Equality, inequality, strict equality, and strict inequality
&	Bitwise AND
^	Bitwise XOR

|	Bitwise OR
&&	Logical AND
||	Logical OR
?:	Conditional
= *OP*=	Assignment, assignment with operation
,	Multiple evaluation

Parentheses are used to alter the order of evaluation determined by operator precedence. This means an expression within parentheses is fully evaluated before its value is used in the remainder of the expression.

For example:

```
z = 78 * (96 + 3 + 45)
```

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

1. Evaluation of the expression within the parentheses occurs first. Within the parentheses, there are two addition operators. Since the addition operators both have the same precedence, they are evaluated from left to right. 96 and 3 are added together first, then 45 is added to this total, resulting in a value of 144.
2. Multiplication occurs next. 78 is multiplied by 144, resulting in a value of 11232.
3. Assignment occurs last. 11232 is assigned to z.

Build: Topic Version 5.6.9309.1546

JScript

Controlling Program Flow

Normally, statements in a JScript script are executed one after the other, in the order in which they are written. This is called sequential

execution, and is the default direction of program flow.

An alternative to sequential execution transfers the program flow to another part of your script. That is, instead of executing the next statement in the sequence, another statement is executed instead.

To make a script useful, this transfer of control must be done in a logical manner. Transfer of program control is based upon a decision, the result of which is a truth statement (returning a Boolean **true** or **false**). You create an expression, then test whether its result is true. There are two main kinds of program structures that accomplish this.

The first is the selection structure. You use it to specify alternate courses of program flow, creating a junction in your program (like a fork in a road). There are four selection structures available in JScript.

- the single-selection structure (**if**),
- the double-selection structure (**if/else**),
- the inline ternary operator **?:**
- the multiple-selection structure (**switch**).

The second type of program control structure is the repetition structure. You use it to specify that an action is to be repeated while some condition remains true. When the conditions of the control statement have been met (usually after some specific number of iterations), control passes to the next statement beyond the repetition structure. There are four repetition structures available in JScript.

- the expression is tested at the top of the loop (**while**),
- the expression is tested at the bottom of the loop (**do/while**),
- operate on each of an object's properties (**for/in**).
- counter controlled repetition (**for**).

You can create quite complex scripts by nesting and stacking selection and repetition control structures.

A third form of structured program flow is provided by exception handling, which is not covered in this document.

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, the relevant JScript code 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 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 (or can be coerced to) **true**, the statement or block of statements after the **if** is executed.

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

// In this example, the test fails unless both conditions are true.
if (rind.color == "deep yellow " && rind.texture == "large and small wrinkles")
{
   theResponse = ("Is it a Crenshaw melon?");
}

// In this example, the test succeeds if either condition is true.
var theReaction = "";
if ((dayOfWeek == "Saturday") || (dayOfWeek == "Sunday"))
{
   theReaction = ("I'm off to the beach!");
}
else
{
   theReaction = ("Hi ho, hi ho, it's off to work I go!");
}
```

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). It also specifies two alternatives, one to be used if the condition is met and one if it is not. A colon must separate these alternatives.

```
var hours = "";

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

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

If you have several conditions to be tested together, and you know that one is more likely to pass or fail than the others, you can use a feature called 'short circuit evaluation' to speed the execution of your script. When JScript evaluates a logical expression, it only evaluates as many

sub-expressions as required to get a result.

For example, if you have andAnd' expression such as ((x == 123) && (y == 42)), JScript first checks if x is 123. If it is not, the entire expression cannot be true, even if y is equal to 42. Hence, the test for y is never made, and JScript returns the value **false**.

Similarly, if only one of several conditions must be **true** (using the || operator), testing stops as soon as any one condition passes the test. This is effective if the conditions to be tested involve the execution of function calls or other complex expressions. With this in mind, when you write Or expressions, place the conditions most likely to be **true** first. When you write And expressions, place the conditions most likely to be **false** first.

A benefit of designing your script in this manner is that **runsecond()** will not be executed in the following example if **runfirst()** returns 0 or **false**.

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

Using Loops

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

Using for Loops

The **for** statement specifies a counter variable, a test condition, and an action that updates the counter. Before each iteration of the loop, the condition is tested. If the test is successful, the code inside the loop is executed. If the test is unsuccessful, the code inside the loop is not executed, and the program continues on the first line of code immediately following the loop. 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. 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 be cautious when writing your loop conditions.

```
/*
The update expression ("icount++" in the following examples)
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 = 10; // Sets a limit of 10 on the loop.

var sum = new Array(howFar); // Creates an array called sum with 10 members, 0 through 9.
var theSum = 0;
sum[0] = 0;

for(var icount = 0; icount < howFar; icount++) { // Counts from 0 through 9 in this case.
theSum += icount;
sum[icount] = theSum;
}

var newSum = 0;
for(var icount = 0; icount > howFar; icount++) { // This isn't executed at all, since icount is not greater than h
newSum += icount;
}

var sum = 0;
for(var icount = 0; 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 user-defined properties of an object, or all the elements of an array. The loop counter in a **for...in** loop is a string, not a number. It contains the name of current property or the index of the current array element.

The following code sample should be run from within Internet Explorer, since it uses the **alert** method, which is not a part of JScript.

```
// Create an object with some properties
var myObject = new Object();
myObject.name = "James";
myObject.age = "22";
myObject.phone = "555 1234";

// Enumerate (loop through)_all the properties in the object
for (prop in myObject)
{
    // This displays "The property 'name' is James", etc.
    window.alert("The property '" + prop + "' is " + myObject[prop]);
}
```

Although **for...in** loops look similar to VBScript's **For Each...Next** loops, they do not work the same way. The JScript **for...in loop** iterates over properties of JScript objects. The VBScript **For Each...Next** loop iterates over items in a collection. To loop over collections in JScript, you need to use the **Enumerator** object. Although some objects, such as those in Internet Explorer, support both VBScript's **For Each...Next** and JScript's **for...in** loops, most objects do not.

Using while Loops

A **while** loop is similar to a **for** loop. The difference is, a **while** loop does not have a built-in counter variable or update expression. If you want to control repetitive execution of a statement or block of statements, but need a more complex rule than simply "run this code n times", use a **while** loop. The following example uses the Internet Explorer object model and a **while** loop to ask the user a simple question.

```
var x = 0;
while ((x != 42) && (x != null))
{
    x = window.prompt("What is my favourite number?", x);
}

if (x == null)
    window.alert("You gave up!");
else
    window.alert("Yep - it's the Ultimate Answer!");
```

> **Note** Because **while** loops do not have explicit built-in counter variables, they are more vulnerable to infinite looping than the other types of loops. Moreover, because it is not necessarily easy to discover where or when the loop condition is updated, it is easy to write a **while** loop in which the condition never gets updated. For this reason, you should be careful when you design **while** loops.

As noted above, there is also a **do...while** loop in JScript that is similar to the while loop, except that it is guaranteed to always execute at least once, since the condition is tested at the end of the loop, rather than at the start. For example, the loop above can be re-written as:

```
var x = 0;
do
{
    x = window.prompt("What is my favourite number?", x);
} while ((x != 42) && (x != null));

if (x == null)
    window.alert("You gave up!");
else
```

```
    window.alert("Yep - it's the Ultimate Answer!");
```

Using break and continue Statements

In Microsoft JScript, the break statement is used to stop the execution of a loop, if some condition is met. (Note that **break** is also used to exit a **switch** block). The continue statement can be used to jump immediately to the next iteration, skipping the rest of the code block, while updating the counter variable if the loop is a **for** or **for...in** loop.

The following example builds on the previous example to use the **break** and **continue** statements to control the loop.

```
var x = 0;
do
{
    x = window.prompt("What is my favourite number?", x);

    // Did the user cancel? If so, break out of the loop
    if (x == null)
        break;

    // Did they enter a number?
    // If so, no need to ask them to enter a number.
    if (Number(x) == x)
        continue;

    // Ask user to only enter in numbers
    window.alert("Please only enter in numbers!");

} while (x != 42)

if (x == null)
    window.alert("You gave up!");
else
    window.alert("Yep - it's the Ultimate Answer!");
```

Build: Topic Version 5.6.9309.1546

JScript

JScript Functions

Microsoft JScript functions perform actions; they can also return values. Sometimes these are the results of calculations or comparisons. Functions are also called "global methods".

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 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 passed to a function are called arguments or parameters. Some functions do not take any arguments at all while others take one or more arguments. In some functions, 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 let you handle expressions and special characters, while others convert strings to numeric values. A useful built-in function is eval(). This function evaluates any valid JScript code that is presented in string form. The **eval ()** function takes one argument, the code to be evaluated. Here is an example using this function.

```
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.
// Assign a string to totality (note the nested quotes)
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 needed. A function definition consists of a function statement and a block of JScript statements.

The **checkTriplet** function in the following example takes the lengths of the sides of a triangle as its arguments. It 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 round-off errors in floating-point calculations, it is not practical to make a direct test of whether the three numbers constitute a Pythagorean triplet 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 = 0.00000000001; // Some very small number to test against.

// The test function for integers.
function integerCheck(a, b, c)
{
   // The test itself.
   if ( (a*a) == ((b*b) + (c*c)) )
      return true;

   return false;
} // End of the integer checking function.

// The test function for floating-point numbers.
function floatCheck(a, b, c)
{
   // Make the test number.
   var delta = ((a*a) - ((b*b) + (c*c)))

   // The test requires the absolute value
   delta = Math.abs(delta);

   // If the difference is less than epsilon, then it's pretty close.
   if (delta < epsilon)
      return true;

   return false;
} // End of the floating-poing check function.


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

```
   // Create a temporary variable for swapping values
   var d = 0;

   // First, move the longest side to position "a".

   // Swap a and b if necessary
   if (b > a)
   {
      d = a;
      a = b;
      b = d;
   }

   // Swap a and c if necessary
   if (c > a)
   {
      d = a;
      a = c;
      c = d;
   }

   // Test all 3 values. Are they integers?
   if (((a % 1) == 0) && ((b % 1) == 0) && ((c % 1) == 0))
   {
      // If so, use the precise check.
      return integerCheck(a, b, c);
   }
   else
   {
      // If not, get as close as is reasonably possible.
      return floatCheck(a, b, c);
   }
} // End of the triplet check function.

// The next three statements assign sample values for testing purposes.
var sideA = 5;
var sideB = 5;
var sideC = Math.sqrt(50.001);

// Call the function. After the call, 'result' contains the result.
var result = checkTriplet(sideA, sideB, sideC);
```

Build: Topic Version 5.6.9309.1546

JScript

JScript Objects

JScript objects are collections of properties and methods. A method is a function that is a member of an object. 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 four kinds of objects: intrinsic objects, objects you create, host objects, which are provided by the host (such as **window** and **document** in Internet Explorer) and Active X objects (external components).

Objects as Arrays

In JScript, objects and arrays are handled almost identically. Both can have arbitrary properties assigned to them, and indeed Arrays are merely a special kind of Object. The difference between Arrays and Objects is that arrays have a "magic" **length** property, whilst objects do not. This means that if you assign a value to an element of an array that is greater than every other element — for example, **myArray[100] = "hello"** — then the **length** property will automatically be updated to be 101 (the new length). Similarly, if you modify the **length** property of an array, it will delete any elements that are no longer part of the array.

All objects in JScript support "expando" properties, or properties that can be added and removed dynamically at run time. These properties can have any name, including numbers. If the name of the property is a simple identifier<<ref for identifier rules>>, it can be written after the object name with a period, such as:

```
var myObj = new Object();

// Add two expando properties, 'name' and 'age'
myObj.name = "Fred";
myObj.age = 42;
```

If the name of the property is not a simple identifier, or it is not known at the time you write the script, you can use an arbitrary expression inside square brackets to index the property. The names of all expando properties in JScript are converted to strings before being added to the object.

```
var myObj = new Object();

// Add two expando properties that cannot be written in the
// object.property syntax.
// The first contains invalid characters (spaces), so must be
// written inside square brackets.
myObj["not a valid identifier"] = "This is the property value";

// The second expando name is a number, so it also must
// be placed inside square brackets
myObj[100] = "100";
```

Traditionally, array elements are given numeric indices, starting at zero. It is these elements that interact with the **length** property. Nevertheless, because all arrays are also objects, they support expando properties as well. Note, though, that expando properties do not interact with the **length** property in any way. For example:

```
// An array with three elements
var myArray = new Array(3);

// Add some data
myArray[0] = "Hello";
myArray[1] = 42;
myArray[2] = new Date(2000, 1, 1);

// This will display 3, the length of the array
window.alert(myArray.length);

// Add some expando properties
myArray.expando = "JScript!";
myArray["another Expando"] = "Windows";

// This will still display 3, since the two expando properties
// don't affect the length.
window.alert(myArray.length);
```

Although JScript does not directly support multi-dimensional arrays, you can store any sort of data inside array elements — including other arrays. So you can get the behavior of multi-dimensional arrays by storing arrays within the elements of another array. For example, the following code builds a multiplication table for the numbers up to 5:

```
// Change this number for a bigger table
var iMaxNum = 5;
// Loop counters
```

```
var i, j;

// New array. Make it iMaxNum + 1 because arrays start
// counting at zero, not 1.
var MultiplicationTable = new Array(iMaxNum + 1);

// Loop for each major number (each row in the table)
for (i = 1; i <= iMaxNum; i++)
{
    // Create the columns in the table
    MultiplicationTable[i] = new Array(iMaxNum + 1);

    // Fill the row with the results of the multiplication
    for (j = 1; j <= iMaxNum; j++)
    {
        MultiplicationTable[i][j] = i * j;
    }
}

window.alert(MultiplicationTable[3][4]); // Displays 12
window.alert(MultiplicationTable[5][2]); // Displays 10
window.alert(MultiplicationTable[1][4]); // Displays 4
```

Build: Topic Version 5.6.9309.1546

JScript

Creating Your Own Objects

To create instances of your own objects, you must first define a constructor function for them. A constructor function creates a new object, giving it properties and, if appropriate, methods. For instance, the following example defines a constructor function for pasta objects. Notice the use of the **this** keyword, which refers to the current object.

```
// pasta is a constructor that takes four parameters.
```

```
function pasta(grain, width, shape, hasEgg)
{
    // What grain is it made of?
    this.grain = grain;

    // How wide is it? (number)
    this.width = width;

    // What is the cross-section? (string)
    this.shape = shape;

    // Does it have egg yolk as a binder? (boolean)
    this.hasEgg = hasEgg;
}
```

Once you define an object constructor, you create instances of it with the **new** operator.

```
var spaghetti = new pasta("wheat", 0.2, "circle", true);
var linguine = new pasta("wheat", 0.3, "oval", true);
```

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 other objects made with the same constructor, 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 constructor function, or to the constructor's prototype object (prototypes are discussed in the Advanced documentation).

```
// Additional properties for spaghetti.
spaghetti.color = "pale straw";
spaghetti.drycook = 7;
spaghetti.freshcook = 0.5;

var chowFun = new pasta("rice", 3, "flat", false);
// Neither the chowFun object, nor any of the other existing
// pasta objects have the three new properties that were added
// to the spaghetti object.


// Adding the 'foodgroup' property to the pasta prototyp object
// makes it available to all instances of pasta objects,
// including those that have already been created.
pasta.prototype.foodgroup = "carbohydrates"

// now spaghetti.foodgroup, chowFun.foodgroup, etc. all
```

```
// contain the value "carbohydrates"
```

Including Methods in the Definition

It is possible to include methods (functions) in the definition of an object. One way to do this is to add include a property in the constructor function that refers to a function defined elsewhere. For instance, the following example expands on the pasta constructor function defined above to include a **toString** method that will be called if you display the value of the object.

```
// pasta is a constructor that takes four parameters.
// The first part is the same as above
function pasta(grain, width, shape, hasEgg)
{
    // What grain is it made of?
    this.grain = grain;

    // How wide is it? (number)
    this.width = width;

    // What is the cross-section? (string)
    this.shape = shape;

    // Does it have egg yolk as a binder? (boolean)
    this.hasEgg = hasEgg;

    // Here we add the toString method (which is defined below).
    // Note that we don't put the parentheses after the name of
    // the function; this is not a function call, but a
    // reference to the function itself.
    this.toString = pastaToString;
}

// The actual function to display the contents of a pasta object.
function pastaToString()
{
    // return the properties of the object

    return "Grain: " + this.grain + "\n" +
        "Width: " + this.width + "\n" +
        "Shape: " + this.shape + "\n" +
        "Egg?: " + Boolean(this.hasEgg);
}
```

```
var spaghetti = new pasta("wheat", 0.2, "circle", true);
// This will call toString() and display the properties
// of the spaghetti object (required Internet Explorer).
window.alert(spaghetti);
```

Build: Topic Version 5.6.9309.1546

JScript

Intrinsic Objects

Microsoft JScript provides eleven intrinsic (or "built-in") objects. They are the **Array**, **Boolean**, **Date**, **Function**, **Global**, **Math**, **Number**, **Object**, **RegExp**, **Error**, and **String** objects. Each of the intrinsic objects has associated methods and properties that are described in detail in the language reference. Certain objects are also described in this section.

Array Object

The subscripts of an array can be thought of as properties of an object, are referred to by their numeric index. Note that named properties added to an Array cannot be indexed by number; they are separate from the array elements.

To create a new array, use the **new** operator and the **Array()** constructor, as in the following example.

```
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 using the **Array** keyword, JScript includes a **length** property, which records the number of entries. 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. In addition, the number of parameters is assigned to the length property, as in the following example, which is equivalent to the preceding example.

```
var theMonths = new Array("Jan", "Feb", "Mar", "Apr", "May", "Jun",
"Jul", "Aug", "Sep", "Oct", "Nov", "Dec");
```

JScript automatically changes the value of **length** when you add elements to an array that you created with the **Array** keyword. Array indices in JScript always start at 0, not 1, so the length property is always one greater than the largest index in the array.

String Object

In JScript, you can treat strings (and numbers) as if they were objects. 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.

```
aString = "0123456789";
var aChunk = aString.substring(4, 7);  // Sets aChunk to "456".
var aNotherChunk = aString.substring(7, 4);  // Sets aNotherChunk to "456".
// Using the preceding Array creation example:
firstLetter = theMonths[5].substring(0,1);  // Sets the firstLetter variable to "J".
```

Another property of the **String** object is the **length** property. This property contains the number of characters in the string (0 for an empty string). This a numeric value, and can be used directly in calculations.

```
var howLong = "Hello World".length  // Sets the howLong variable to 11.
```

Math Object

The **Math** object has a number of predefined properties and methods. The properties are specific numbers. One of these specific numbers 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 value.
var circleArea = Math.PI * radius * radius;  // Note capitalization of Math and PI.
```

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 uses both pi and exponentiation.

```
// This formula calculates the volume of a sphere with the given radius.
volume = (4/3)*(Math.PI*Math.pow(radius,3));
```

Date Object

The **Date** object can be used to represent arbitrary dates and times, to get the current system date, and to calculate differences between dates. It has several 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, which is Greenwich Mean Time (the preferred term is UTC, or "Universal Coordinated Time," which refers to signals issued by the World Time Standard). JScript can handle dates that are in the approximate range 250,000 B.C. to 255,000 A.D.

To create a new **Date** object, 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.

```
/*
This example uses the array of month names defined previously.
The first statement assigns today's date, in "Day Month Date 00:00:00 Year"
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.
var thisYear = toDay.getFullYear();
var thisMonth = theMonths[toDay.getMonth()];
var thisDay = thisMonth  + " " + toDay.getDate() + ", " + thisYear;
```

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.

Property	Description
MAX_VALUE	Largest possible number, about 1.79E+308; can be positive or negative. (Value varies slightly

	from system to system.)
MIN_VALUE	Smallest possible number, about 2.22E-308; can be positive or negative. (Value varies slightly from system to system.)
NaN	Special nonnumeric value, "not a number."
POSITIVE_INFINITY	Any positive value larger than the largest positive number (Number.MAX_VALUE) is automatically converted to this value; represented as infinity.
NEGATIVE_INFINITY	Any value more negative than the largest negative number (-Number.MAX_VALUE) is automatically converted to this value; represented as -infinity.

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 to itself. To test for a **NaN** result, do not compare against **Number.NaN**; use the **isNaN()** function instead.

Build: Topic Version 5.6.9309.1546

JScript

JScript Reserved Words

JScript has a number of reserved words that you cannot use as identifiers. Reserved words have a specific meaning to the JScript language, as they are part of the language syntax. Using a reserved word causes a compilation error when loading your script.

JScript also has a list of future reserved words. These words are not currently part of the JScript language, although they are reserved for future use.

Reserved Words

break	delete	function	return	typeof
case	do	if	switch	var
catch	else	in	this	void

continue	false	instanceof	throw	while
debugger	finally	new	true	with
default	for	null	try	

Future Reserved Words

abstract	double	goto	native	static
boolean	enum	implements	package	super
byte	export	import	private	synchronized
char	extends	int	protected	throws
class	final	interface	public	transient
const	float	long	short	volatile

When choosing identifiers it is also important to avoid any words that are already the names of intrinsic JScript objects or functions, such as **String** or **parseInt**.

Build: Topic Version 5.6.9309.1546

JScript

Advanced JScript

Advanced Object Creation

Recursion

Variable Scope

Copying, Passing, and Comparing Data

Using Arrays

Special Characters

Troubleshooting Your Scripts

Build: Topic Version 5.6.9309.1546

JScript

Advanced Object Creation

A constructor is a function you call to instantiate and initialize a particular type of object. You invoke a constructor with the **new** keyword. Here are a few examples of using constructors.

```
var myObject = new Object();             // Creates a generic object with no properties.
var myBirthday = new Date(1961, 5, 10);  // Creates a Date object.
var myCar = new Car();                   // Creates a user defined object, and initializes its properties.
```

The constructor is passed a reference to a newly created empty object as the value of the special **this** keyword. It is then responsible for performing appropriate initialization for the new object (creating properties and giving them initial values). When completed, the constructor returns a reference to the object it constructed.

Writing Constructors

You can create objects and initialize them using the **new** operator in conjunction with predefined constructor functions such as **Object()**, **Date()**, and **Function()**. A powerful feature of object-oriented programming is the ability to define custom constructor functions to create custom objects for use in your scripts. You create custom constructors so you can create objects with properties already defined. Here is an example of a custom constructor (note the use of the **this** keyword).

```
function Circle (xPoint, yPoint, radius) {
```

```
    this.x = xPoint;  // The x component of the center of the circle.
    this.y = yPoint;  // The y component of the center of the circle.
    this.r = radius;  // The radius of the circle.
}
```

When you invoke the Circle constructor, you supply values for the circle's center point and the radius (these elements are all that is needed to completely define a unique circle object). You end up with a Circle object that contains three properties. Here is how you would instantiate a Circle object.

```
var aCircle = new Circle(5, 11, 99);
```

Using Prototypes to Create Objects

When you write a constructor, you can use properties of the prototype object (which is itself a property of every constructor) to create inherited properties, and shared methods. Prototype properties and methods are copied by reference into each object of a class, so they all have the same values. 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. Here is an example that makes use of the custom constructor, Circle (note the use of the **this** keyword).

```
Circle.prototype.pi = Math.PI;
function ACirclesArea () {
    return this.pi * this.r * this.r; // The formula for the area of a circle is Ïr².
}
Circle.prototype.area = ACirclesArea; // The function that calculates the area of a circle is now a method of the
var a = ACircle.area();               // This is how you would invoke the area function on a Circle object.
```

Using this principle, you can define additional properties for predefined constructor functions (which all have prototype objects). For example, if you want to be able to remove leading and trailing spaces from strings (similar to VBScript's **Trim** function), you can create your own method on the **String** prototype object, and all strings in your script will automatically inherit the method.

```
// Add a function called trim as a method of the prototype
// object of the String constructor.
String.prototype.trim = function()
{
    // Use a regular expression to replace leading and trailing
    // spaces with the empty string
    return this.replace(/(^\s*)|(\s*$)/g, "");
}
```

```
// A string with spaces in it
var s = "    leading and trailing spaces    ";

// Displays "    leading and trailing spaces     (35)"
window.alert(s + " (" + s.length + ")");

// Remove the leading and trailing spaces
s = s.trim();
// Displays "leading and trailing spaces (27)"
window.alert(s + " (" + s.length + ")");
```

Build: Topic Version 5.6.9309.1546

JScript

Recursion

Recursion is an important programming technique. It is used to have a function call itself from within itself. One example is the calculation of factorials. The factorial of 0 is 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 are 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 is not an integer, round it down to the next integer. If the number is zero, its factorial is one. If the number is larger than zero, multiply it by the factorial of the next lesser number."

To calculate the factorial of any number that is larger than zero, 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.

Recursion and iteration (looping) are strongly related - anything that can be done with recursion can be done with iteration, and vice-versa. Usually a particular computation will lend itself to one technique or the other, and you simply need to choose the most natural approach, or

the one you feel most comfortable with.

Clearly, there is a way to get in trouble here. You can easily create a recursive function that does not 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 is 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 (whatever number you decide that should be) it automatically quits.

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

```
// Function to calculate factorials. If an invalid
// number is passed in (ie, one less than zero), -1
// is returned to signify an error. Otherwise, the
// number is converted to the nearest integer, and its
// factorial is returned.
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 -1;
    }
      if (aNumber == 0)  {  // If the number is 0, its factorial is 1.
      return 1;
      }
        else return (aNumber * factorial(aNumber - 1));  // Otherwise, recurse until done.
}
```

Build: Topic Version 5.6.9309.1546

Variable Scope

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.

Languages such as C++ also have "block scope." Here, any set of braces "{}" defines a new scope. JScript does not support block scopes.

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.

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

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

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

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

// 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.

```
tweak();
var aNumber = 100;
```

```
function tweak()  {
    var newThing = 0;  // Explicit declaration of the newThing variable.

    // This statement assigns the value undefined to newThing because there is a local variable with the name aNum
    newThing = aNumber;

    // The next statement assigns the value 42 to the local aNumberaNumber = 42;
    if (false)  {
        var aNumber;  // This statement is never executed.
        aNumber = 123;  // This statement is never executed.
        }  // End of the conditional.

}  // End of the function definition.
```

When JScript executes a function, it first looks for all variable declarations,

```
var someVariable;
```

and creates the variables with an initial value of undefined. If a variable is declared with a value,

```
var someVariable = "something";
```

then it still initially has the value undefined, and will take on the declared value only when the line containing the declaration is executed, if ever.

JScript processes variable declarations before executing any code, so it does not matter whether the declaration is inside a conditional block or some other construct. Once JScript has found all the variables, it executes the code in the function. If a variable is implicitly declared inside a function - that is, if it appears on the left-hand-side of an assignment expression but has not been declared with var - then it is created as a global variable.

Build: Topic Version 5.6.9309.1546

JScript

Copying, Passing, and Comparing Data

In 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 copy 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*. 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.

When comparing by reference, the two variables must refer to exactly the same entity for the comparison to succeed. For example, two distinct **Array** objects will always compare as unequal, even if they contain the same elements. One of the variables must be a reference to the other one for the comparison to succeed. To check if two Arrays hold the same elements, compare the results of the **toString()** method.

Last, strings are copied and passed by reference, but are compared by value. Note that if you have two **String** objects (created with **new** String("something")), they are compared by reference, but if one or both of the values is a string value, they 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." You can call **toUpperCase()** or **toLowerCase()** on both strings if you want to perform a case-insensitive match.

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. Even though objects and arrays are passed by reference, if you directly overwrite them with a new value in the function, the new value is not reflected outside the function. Only changes to properties of objects, or elements of arrays, are visible outside the function.

For example (using the Internet Explorer object model):

```
// This clobbers (over-writes) its parameter, so the change
// is not reflected in the calling code.
```

```
function Clobber(param)
{
    // clobber the parameter; this will not be seen in
    // the calling code
    param = new Object();
    param.message = "This will not work";
}

// This modifies a property of the parameter, which
// can be seen in the calling code.
function Update(param)
{
    // Modify the property of the object; this will be seen
    // in the calling code.
    param.message = "I was changed";
}

// Create an object, and give it a property.
var obj = new Object();
obj.message = "This is the original";

// Call Clobber, and print obj.message. Note that it hasn't changed.
Clobber(obj);
window.alert(obj.message); // Still displays "This is the original".

// Call Update, and print obj.message. Note that is has changed.
Update(obj);
window.alert(obj.message); // Displays "I was changed".
```

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.

Build: Topic Version 5.6.9309.1546

JScript

Using Arrays

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 are not required to.

In JScript, objects and arrays are almost identical to each other. The two main differences are that normal objects do not have an automatic length property, and arrays do not have the properties and methods of an object.

Addressing Arrays

You address arrays by using brackets "[]". The brackets enclose either a numeric value, or an expression that evaluates to a whole number. The following example assumes that the *entryNum* variable is defined and assigned a value elsewhere in the script.

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

Objects as Associative Arrays

Normally, you use the dot operator "." to access an object's properties. For example,

```
myObject.aProperty
```

Here, the property name is an identifier. You can also access an object's properties using the index operator "[]". Here, you are treating the object as an *associative array*. An associative array is a data structure that allows you to dynamically associate arbitrary data values with arbitrary strings. For example,

```
myObject["aProperty"] // Same as above.
```

Although the use of the index operator is more commonly associated with accessing array elements, when used with objects, the index is always the property name expressed as a string literal.

Notice the important difference between the two ways of accessing object properties.

Operator	The property name is treated as	Meaning the property name
Dot "."	an identifier	cannot be manipulated as data
Index "[]"	a string literal	can be manipulated as data

This difference becomes useful when you do not know what the property names will be until runtime (for example, when you are constructing objects based on user input). To extract all the properties from an associative array, you must use the **for … in** loop.

Build: Topic Version 5.6.9309.1546

JScript

Special Characters

JScript provides special characters that allow you to include in strings some characters you cannot 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.

Escape Sequence	Character
\b	Backspace
\f	Form feed
\n	Line feed (newline)
\r	Carriage return
\t	Horizontal tab (Ctrl-I)
\'	Single quotation mark
\"	Double quotation mark

\\ Backslash

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 (\\).

```
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."');
```

Build: Topic Version 5.6.9309.1546

JScript

Troubleshooting Your Scripts

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 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.

Note This behavior is specific to Internet Explorer. ASP and WSH have different execution models (as would other hosts).

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;
```

To check that both the type and value are the same, use the strict equality operator, ===. the following both evaluate to false:

```
"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 * theCorrectionFactor;
```

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 in Internet Explorer

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 in Internet Explorer

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.

Build: Topic Version 5.6.9309.1546

JScript

Conditional Compilation

Conditional compilation allows the use of new JScript language features without sacrificing compatibility with older versions that do not support the features.

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

Always place conditional compilation code in comments, so that hosts (like Netscape Navigator) that do not understand conditional compilation will ignore it. Here is an example.

```
/*@cc_on @*/
```

```
/*@if (@_jscript_version >= 4)
   alert("JScript version 4 or better");
   @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 support conditional compilation only see the message informing of the need for a new scripting engine.

Build: Topic Version 5.6.9309.1546

JScript

Conditional Compilation Variables

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.

Variable	Description
@_win32	True if running on a Win32 system.
@_win16	True if running on a Win16 system.
@_mac	True if running on an Apple Macintosh system.
@_alpha	True if running on a DEC Alpha processor.
@_x86	True if running on an Intel processor.
@_mc680x0	True if running on a Motorola 680x0 processor.
@_PowerPC	True if running on a Motorola PowerPC processor.
@_jscript	Always true.
@_jscript_build	Contains the build number of the JScript scripting engine.
@_jscript_version	Contains the JScript version number in major.minor format.

Build: Topic Version 5.6.9309.1546

JScript

Introduction to Regular Expressions

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

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

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

Build: Topic Version 5.6.9309.1546

JScript

Regular Expressions

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

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

data1.dat

data2.dat

datax.dat

dataN.dat

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

data.dat

data1.dat

data2.dat

data12.dat

datax.dat

dataXYZ.dat

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

Build: Topic Version 5.6.9309.1546

JScript

Early Beginnings

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

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

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

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

Build: Topic Version 5.6.9309.1546

JScript

Uses for Regular Expressions

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

With regular expressions, you can:

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

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

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

Build: Topic Version 5.6.9309.1546

JScript

Regular Expression Syntax

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

Here are some examples of regular expression you might encounter:

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

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

Character	Description
\	Marks the next character as either a special character, a literal, a backreference, or an octal escape. For example, 'n' matches the character "n". '\n' matches a newline character. The sequence '\\' matches "\" and "\(" matches "(".
^	Matches the position at the beginning of the input string. If the **RegExp** object's **Multiline** property is set, ^ also matches the position following '\n' or '\r'.
$	Matches the position at the end of the input string. If the **RegExp** object's **Multiline** property is set, $ also matches the position preceding '\n' or '\r'.
*	Matches the preceding subexpression zero or more times. For example, zo* matches "z" and "zoo". * is equivalent to {0,}.
+	Matches the preceding subexpression one or more times. For example, 'zo+' matches "zo" and "zoo", but not "z". + is

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

[^*a-z*]	A negative range characters. Matches any character not in the specified range. For example, '[^a-z]' matches any character not in the range 'a' through 'z'.
\b	Matches a word boundary, that is, the position between a word and a space. For example, 'er\b' matches the 'er' in "never" but not the 'er' in "verb".
\B	Matches a nonword boundary. 'er\B' matches the 'er' in "verb" but not the 'er' in "never".
\c*x*	Matches the control character indicated by *x*. For example, \cM matches a Control-M or carriage return character. The value of *x* must be in the range of A-Z or a-z. If not, c is assumed to be a literal 'c' character.
\d	Matches a digit character. Equivalent to [0-9].
\D	Matches a nondigit character. Equivalent to [^0-9].
\f	Matches a form-feed character. Equivalent to \x0c and \cL.
\n	Matches a newline character. Equivalent to \x0a and \cJ.
\r	Matches a carriage return character. Equivalent to \x0d and \cM.
\s	Matches any whitespace character including space, tab, form-feed, etc. Equivalent to [\f\n\r\t\v].
\S	Matches any non-whitespace character. Equivalent to [^ \f\n\r\t\v].
\t	Matches a tab character. Equivalent to \x09 and \cI.
\v	Matches a vertical tab character. Equivalent to \x0b and \cK.
\w	Matches any word character including underscore. Equivalent to '[A-Za-z0-9_]'.
\W	Matches any nonword character. Equivalent to '[^A-Za-z0-9_]'.
\x*n*	Matches *n*, where *n* is a hexadecimal escape value. Hexadecimal escape values must be exactly two digits long. For example, '\x41' matches "A". '\x041' is equivalent to '\x04' & "1". Allows ASCII codes to be used in regular expressions.
num	Matches *num*, where *num* is a positive integer. A reference back to captured matches. For example, '(.)\1' matches two consecutive identical characters.
n	Identifies either an octal escape value or a backreference. If *n* is preceded by at least *n* captured subexpressions, *n* is a backreference. Otherwise, *n* is an octal escape value if *n* is an octal digit (0-7).
nm	Identifies either an octal escape value or a backreference. If *nm* is preceded by at least *nm* captured subexpressions, *nm* is a backreference. If *nm* is preceded by at least *n* captures, *n* is a backreference followed by literal *m*. If neither of the preceding conditions exist, *nm* matches octal escape value *nm* when *n* and *m* are octal digits (0-7).
nml	Matches octal escape value *nml* when *n* is an octal digit (0-3) and *m* and *l* are octal digits (0-7).
\u*n*	Matches *n*, where *n* is a Unicode character expressed as four hexadecimal digits. For example, \u00A9 matches the copyright symbol (©).

Build: Topic Version 5.6.9309.1546

JScript

Build a Regular Expression

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

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

```
/expression/
```

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

```
"expression"
```

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

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

Build: Topic Version 5.6.9309.1546

JScript

Order of Precedence

Once you have constructed a regular expression, it is evaluated much like an arithmetic expression, that is, it is evaluated from left to right

and follows an order of precedence.

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

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

Build: Topic Version 5.6.9309.1546

JScript

Ordinary Characters

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

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

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

The equivalent VBScript single-character regular expressions are:

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

```
"M"
```

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

```
/a7M/
```

The equivalent VBScript expression is:

```
"a7M"
```

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

Build: Topic Version 5.6.9309.1546

JScript

Special Characters

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

Special Character	Comment
$	Matches the position at the end of an input string. If the **RegExp** object's **Multiline** property is set, $ also matches the position preceding '\n' or '\r'. To match the $ character itself, use \$.
( )	Marks the beginning and end of a subexpression. Subexpressions can be captured for later use. To match these characters, use \(and \).
*	Matches the preceding subexpression zero or more times. To match the * character, use *.
+	Matches the preceding subexpression one or more times. To match the + character, use \+.

.	Matches any single character except the newline character \n. To match ., use \.
[	Marks the beginning of a bracket expression. To match [, use \[.
?	Matches the preceding subexpression zero or one time, or indicates a non-greedy quantifier. To match the ? character, use \?.
\	Marks the next character as either a special character, a literal, a backreference, or an octal escape. For example, 'n' matches the character 'n'. '\n' matches a newline character. The sequence '\\' matches "\" and '\(' matches "(".
^	Matches the position at the beginning of an input string except when used in a bracket expression where it negates the character set. To match the ^ character itself, use \^.
{	Marks the beginning of a quantifier expression. To match {, use \{.
|	Indicates a choice between two items. To match |, use \\|.

Build: Topic Version 5.6.9309.1546

JScript

Non-Printable Characters

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

Character	Meaning
\c*x*	Matches the control character indicated by *x*. For example, \cM matches a Control-M or carriage return character. The value of *x* must be in the range of A-Z or a-z. If not, c is assumed to be a literal 'c' character.
\f	Matches a form-feed character. Equivalent to \x0c and \cL.
\n	Matches a newline character. Equivalent to \x0a and \cJ.
\r	Matches a carriage return character. Equivalent to \x0d and \cM.
\s	Matches any whitespace character including space, tab, form-feed, etc. Equivalent to [\f\n\r\t\v].
\S	Matches any non-whitespace character. Equivalent to [^ \f\n\r\t\v].
\t	Matches a tab character. Equivalent to \x09 and \cI.
\v	Matches a vertical tab character. Equivalent to \x0b and \cK.

Build: Topic Version 5.6.9309.1546

JScript

Character Matching

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

```
/a.c/
```

The equivalent VBScript regular expression is:

```
"a.c"
```

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

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

For VBScript, the equivalent expression appears as follows:

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

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

Bracket Expressions

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

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

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

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

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

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

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

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

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

The same expression for VBScript appears as follows:

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

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

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

- Escape it with a backslash:

  ```
  [\-]
  ```

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

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

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

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

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

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

For VBScript use:

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

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

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

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

or for VBScript:

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

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

JScript expression specifies such a match:

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

The equivalent expression for VBScript is:

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

Build: Topic Version 5.6.9309.1546

JScript

Quantifiers

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

The following table illustrates the various quantifiers and their meanings:

Character	Description
*	Matches the preceding subexpression zero or more times. For example, 'zo*' matches "z" and "zoo". * is equivalent to {0,}.
+	Matches the preceding subexpression one or more times. For example, 'zo+' matches "zo" and "zoo", but not "z". + is equivalent to {1,}.
?	Matches the preceding subexpression zero or one time. For example, 'do(es)?' matches the "do" in "do" or "does". ? is equivalent to {0,1}
{*n*}	*n* is a nonnegative integer. Matches exactly *n* times. For example, 'o{2}' does not match the 'o' in "Bob," but matches the two o's in "food".
{*n*,}	*n* is a nonnegative integer. Matches at least *n* times. For example, 'o{2,}' does not match the 'o' in "Bob" and matches

	all the o's in "fooooood". 'o{1,}' is equivalent to 'o+'. 'o{0,}' is equivalent to 'o*'.
{*n*,*m*}	*m* and *n* are nonnegative integers, where *n* <= *m*. Matches at least *n* and at most *m* times. For example, 'o{1,3}' matches the first three o's in "fooooood". 'o{0,1}' is equivalent to 'o?'. Note that you cannot put a space between the comma and the numbers.

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

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

The following VBScript regular expression performs the identical match:

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

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

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

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

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

For VBScript, use the following regular expression:

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

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

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

-or-

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

For VBScript, the following expressions are equivalent:

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

-or-

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

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

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

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

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

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

The VBScript regular expression is:

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

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

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

-or-

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

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

Build: Topic Version 5.6.9309.1546

JScript

Anchors

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

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

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

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

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

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

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

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

For VBScript the same regular expressions appears as:

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

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

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

For VBScript use:

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

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

```
/\bCha/
```

or for VBScript:

```
"\bCha"
```

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

```
/ter\b/
```

and

```
"ter\b"
```

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

```
/\Bapt/
```

and

```
"\Bapt"
```

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

Build: Topic Version 5.6.9309.1546

JScript

Alternation and Grouping

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

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

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

You can use parentheses to limit the scope of the alternation, that is, make sure that it applies only to the two words, 'Chapter' and 'Section'.

However, parentheses are tricky as well, because they are also used to create subexpressions, something that's covered later in the section on subexpressions. By taking the regular expressions shown above and adding parentheses in the appropriate places, you can make the regular expression match either 'Chapter 1' or 'Section 3'.

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

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

For VBScript:

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

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

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

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

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

For VBScript:

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

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

For example, suppose you have a document containing references to Windows 3.1, Windows 95, Windows 98, and Windows NT. Suppose

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

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

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

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

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

Build: Topic Version 5.6.9309.1546

JScript

Backreferences

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

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

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

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

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

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

The equivalent VBScript expression is:

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

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

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

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

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

The closest equivalent VBScript code appears as follows:

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

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

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

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

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

The following regular expressions provides that functionality. For JScript:

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

For VBScript:

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

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

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

- **RegExp.$1** contains "http"
- **RegExp.$2** contains "msdn.microsoft.com"
- **RegExp.$3** contains ":80"
- **RegExp.$4** contains "/scripting/default.htm"

Build: Topic Version 5.6.9309.1546

JScript

JScript Language Reference

Build: Topic Version 5.6.9309.1546

JScript

Feature Information

The following table lists JScript features.

Description	Language Element
List of JScript versions by host application and list of features by version.	Version Information
List of ECMA features currently in JScript.	JScript Features (ECMA)
List of non-ECMA features currently in JScript.	JScript Features (Non-ECMA)

Build: Topic Version 5.6.9309.1546

JScript

Microsoft JScript Features - ECMA

The following table lists JScript features compliant with ECMA standards.

Category	Feature/Keyword
Array Handling	Array join, length, reverse, sort
Assignments	Assign (=) Compound Assign (OP=)
Booleans	Boolean
Comments	/*...*/ or //
Constants/Literals	NaN null true, false Infinity undefined
Control flow	Break continue for for...in if...else

	return
	while
Dates and Time	Date
	getDate, getDay, getFullYear, getHours, getMilliseconds, getMinutes, getMonth, getSeconds, getTime, getTimezoneOffset, getYear, getUTCDate, getUTCDay, getUTCFullYear, getUTCHours, getUTCMilliseconds, getUTCMinutes, getUTCMonth, getUTCSeconds, setDate, setFullYear, setHours, setMilliseconds, setMinutes, setMonth, setSeconds, setTime, setYear, setUTCDate, setUTCFullYear, setUTCHours, setUTCMilliseconds, setUTCMinutes, setUTCMonth, setUTCSeconds, toGMTString, toLocaleString, toUTCString, parse, UTC
Declarations	Function
	new
	this
	var
	with
Function Creation	Function
	arguments, length
Global Methods	Global
	escape, unescape
	eval
	isFinite, isNaN
	parseInt, parseFloat
Math	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
Numbers	Number
	MAX_VALUE, MIN_VALUE
	NaN
	NEGATIVE_INFINITY, POSITIVE_INFINITY
Object Creation	Object
	new
	constructor, prototype, instanceof, toString, valueOf
Operators	Addition (+), Subtraction (-)

	Modulus arithmetic (%) Multiplication (*), Division (/) Negation (-) Equality (==), Inequality (!=) Less Than (<), Less Than or Equal To (<=) Greater Than (>) Greater Than or Equal To (>=) Logical And(&&), Or (||), Not (!) Bitwise And (&), Or (|), Not (~), Xor (^) Bitwise Left Shift (<<), Shift Right (>>) Unsigned Shift Right (>>>) Conditional (?:) Comma (,) delete, typeof, void Decrement (—), Increment (++)
Objects	Array Boolean Date Function Global Math Number Object String
Strings	String charAt, charCodeAt, fromCharCode indexOf, lastIndexOf split toLowerCase, toUpperCase length

Build: Topic Version 5.6.9309.1546

JScript

Microsoft JScript Features - Non-ECMA

The following table lists JScript features that are not compliant with ECMA standards.

Category	Feature/Keyword
Array Handling	concat, slice VBArray dimensions, getItem, lbound, toArray, ubound
Conditional Compilation	@cc_on @if Statement @set Statement Conditional Compilation Variables
Control flow	do...while Labeled switch
Dates and Time	getVarDate
Enumeration	Enumerator atEnd, item, moveFirst, moveNext
Error Handling	Error description, number throw, try...catch
Function Creation	caller
Operators	Identity (===), Nonidentity (!==)
Objects	Enumerator RegExp Regular Expression VBArray ActiveXObject GetObject
Regular Expressions and Pattern Matching	RegExp index, input, lastIndex, $1...$9, source, compile, exec, test Regular Expression Syntax
Script Engine Identification	ScriptEngine

	ScriptEngineBuildVersion ScriptEngineMajorVersion ScriptEngineMinorVersion
Strings	concat, slice match, replace, search anchor, big, blink, bold, fixed, fontcolor, fontsize, italics, link, small, strike, sub, sup

---

Build: Topic Version 5.6.9309.1546

JScript

JScript Errors

The following table lists the types of JScript errors.

For more information about	See
List of JScript run-time errors	Run-time Errors
List of JScript syntax errors	Syntax Errors

Build: Topic Version 5.6.9309.1546

JScript

JScript Run-time Errors

JScript run-time errors are errors that result when your JScript script attempts to perform an action that the system cannot execute. JScript

run-time errors occur while your script is being executed; when variable expressions are being evaluated, and memory is being dynamic allocated.

Error Number	Description
5029	Array length must be a finite positive integer
5030	Array length must be assigned a finite positive number
5028	Array or arguments object expected
5010	Boolean expected
5003	Cannot assign to a function result
5000	Cannot assign to 'this'
5006	Date object expected
5015	Enumerator object expected
5022	Exception thrown and not caught
5020	Expected ')' in regular expression
5019	Expected ']' in regular expression
5023	Function does not have a valid prototype object
5002	Function expected
5008	Illegal assignment
5021	Invalid range in character set
5014	JScript object expected
5001	Number expected
5007	Object expected
5012	Object member expected
5016	Regular Expression object expected
5005	String expected
5017	Syntax error in regular expression
5026	The number of fractional digits is out of range
5027	The precision is out of range
5025	The URI to be decoded is not a valid encoding
5024	The URI to be encoded contains an invalid character
5009	Undefined identifier
5018	Unexpected quantifier
5013	VBArray expected

See Also

JScript Syntax Errors

Build: Topic Version 5.6.9309.1546

JScript

JScript Syntax Errors

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

Error Number	Description
1019	Can't have 'break' outside of loop
1020	Can't have 'continue' outside of loop
1030	Conditional compilation is turned off
1027	'default' can only appear once in a 'switch' statement
1005	Expected '('
1006	Expected ')'
1012	Expected '/'
1003	Expected ':'
1004	Expected ';'
1032	Expected '@'
1029	Expected '@end'
1007	Expected ']'
1008	Expected '{'
1009	Expected '}'

1011	Expected '='
1033	Expected 'catch'
1031	Expected constant
1023	Expected hexadecimal digit
1010	Expected identifier
1028	Expected identifier, string or number
1024	Expected 'while'
1014	Invalid character
1026	Label not found
1025	Label redefined
1018	'return' statement outside of function
1002	Syntax error
1035	Throw must be followed by an expression on the same source line
1016	Unterminated comment
1015	Unterminated string constant

See Also

JScript Run-time Errors

Build: Topic Version 5.6.9309.1546

JScript

JScript Functions

The following table lists JScript functions.

Description	Language Element

Returns a reference to an Automation object from a file.	GetObject Function
Returns a string representing the scripting language in use.	ScriptEngine Function
Returns the build version number of the scripting engine in use.	ScriptEngineBuildVersion Function
Returns the major version number of the scripting engine in use.	ScriptEngineMajorVersion Function
Returns the minor version number of the scripting engine in use.	ScriptEngineMinorVersion Function

---

Build: Topic Version 5.6.9309.1546

JScript

GetObject Function

Returns a reference to an Automation object from a file.

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

Arguments

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

class
Optional. Class of the object.

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

appname
Required. Name of the application providing the object.

objectype
Required. Type or class of object to create.

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:

```
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. To do so, 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`:

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

If you don not specify the object's class, Automation determines which application to start and which 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:\\DRAWINGS\\SAMPLE.DRW", "FIGMENT.DRAWING");
```

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

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

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

Requirements

Version 5

See Also

ActiveXObject Object

Build: Topic Version 5.6.9309.1546

JScript

ScriptEngine Function

Returns a string representing the scripting language in use.

```
ScriptEngine( )
```

Remarks

The **ScriptEngine** function can return any of the following strings:

String	Description
JScript	Indicates that Microsoft JScript is the current scripting engine.
VBA	Indicates that Microsoft Visual Basic for Applications is the current scripting engine.
VBScript	Indicates that Microsoft Visual Basic Scripting Edition is the current scripting engine.

Example

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

```
function GetScriptEngineInfo(){
   var s;
   s = ""; // Build string with necessary info.
   s += ScriptEngine() + " Version ";
   s += ScriptEngineMajorVersion() + ".";
   s += ScriptEngineMinorVersion() + ".";
   s += ScriptEngineBuildVersion();
   return(s);
}
```

Requirements

Version 5

See Also

ScriptEngineBuildVersion Function | ScriptEngineMajorVersion Function | ScriptEngineMinorVersion Function

---

Build: Topic Version 5.6.9309.1546

JScript

ScriptEngineBuildVersion Function

Returns the build version number of the scripting engine in use.

ScriptEngineBuildVersion( )

Remarks

The return value corresponds directly to the version information contained in the dynamic-link library (DLL) for the scripting language in use.

Example

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

```
function GetScriptEngineInfo(){
   var s;
   s = ""; // Build string with necessary info.
   s += ScriptEngine() + " Version ";
   s += ScriptEngineMajorVersion() + ".";
   s += ScriptEngineMinorVersion() + ".";
   s += ScriptEngineBuildVersion();
   return(s);
}
```

Requirements

Version 5

See Also

ScriptEngine Function | ScriptEngineMajorVersion Function | ScriptEngineMinorVersion Function

---

Build: Topic Version 5.6.9309.1546

JScript

ScriptEngineMajorVersion Function

Returns the major version number of the scripting engine in use.

ScriptEngineMajorVersion( )

Remarks

The return value corresponds directly to the version information contained in the dynamic-link library (DLL) for the scripting language in use.

Example

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

```
function GetScriptEngineInfo(){
   var s;
   s = ""; // Build string with necessary info.
   s += ScriptEngine() + " Version ";
   s += ScriptEngineMajorVersion() + ".";
   s += ScriptEngineMinorVersion() + ".";
   s += ScriptEngineBuildVersion();
   return(s);
}
```

Requirements

Version 5

See Also

ScriptEngine Function | ScriptEngineBuildVersion Function | ScriptEngineMinorVersion Function

---

Build: Topic Version 5.6.9309.1546

JScript

ScriptEngineMinorVersion Function

Returns the minor version number of the scripting engine in use.

```
ScriptEngineMinorVersion( )
```

Remarks

The return value corresponds directly to the version information contained in the dynamic-link library (DLL) for the scripting language in use.

Example

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

```
function GetScriptEngineInfo(){
   var s;
   s = ""; // Build string with necessary info.
   s += ScriptEngine() + " Version ";
   s += ScriptEngineMajorVersion() + ".";
   s += ScriptEngineMinorVersion() + ".";
   s += ScriptEngineBuildVersion();
   return(s);
}
```

Requirements

Version 5

See Also

ScriptEngine Function | ScriptEngineBuildVersion Function | ScriptEngineMajorVersion Function

Build: Topic Version 5.6.9309.1546

JScript

JScript Methods

The following table list JScript Methods

Description	Language Element
Returns the absolute value of a number.	abs Method
Returns the arccosine of a number.	acos Method
Places an HTML anchor with a NAME attribute around specified text in the object.	anchor Method
Returns the arcsine of a number.	asin Method
Returns the arctangent of a number.	atan Method
Returns the angle (in radians) from the X axis to a point (y,x).	atan2 Method
Returns a Boolean value indicating if the enumerator is at the end of the collection.	atEnd Method
Places HTML <BIG> tags around text in a **String** object.	big Method
Places HTML <BLINK> tags around text in a **String** object.	blink Method
Places HTML <B> tags around text in a **String** object.	bold Method
Returns the smallest integer greater than or equal to its numeric argument.	ceil Method
Returns the character at the specified index.	charAt Method
Returns the Unicode encoding of the specified character.	charCodeAt Method
Compiles a regular expression into an internal format.	compile Method
Returns a new array consisting of a combination of two arrays.	concat Method (Array)
Returns a **String** object containing the concatenation of two supplied strings.	concat Method (String)
Returns the cosine of a number.	cos Method
Returns the number of dimensions in a VBArray.	dimensions Method
Encodes **String** objects so they can be read on all computers.	escape Method

Evaluates JScript code and executes it.	eval Method
Executes a search for a match in a specified string.	exec Method
Returns *e* (the base of natural logarithms) raised to a power.	exp Method
Places HTML <TT> tags around text in a **String** object.	fixed Method
Returns the greatest integer less than or equal to its numeric argument.	floor Method
Places an HTML <FONT> tag with the COLOR attribute around the text in a **String** object.	fontcolor Method
Places an HTML <FONT> tag with the SIZE attribute around the text in a **String** object.	fontsize Method
Returns a string from a number of Unicode character values.	fromCharCode Method
Returns the day of the month value in a **Date** object using local time.	getDate Method
Returns the day of the week value in a **Date** object using local time.	getDay Method
Returns the year value in the **Date** object using local time.	getFullYear Method
Returns the hours value in a **Date** object using local time.	getHours Method
Returns the item at the specified location.	getItem Method
Returns the milliseconds value in a **Date** object using local time.	getMilliseconds Method
Returns the minutes value stored in a **Date** object using local time.	getMinutes Method
Returns the month value in the **Date** object using local time.	getMonth Method
Returns seconds value stored in a **Date** object using local time.	getSeconds Method
Returns the time value in a **Date** object.	getTime Method
Returns the difference in minutes between the time on the host computer and Universal Coordinated Time (UTC).	getTimezoneOffset Method
Returns the date value in a **Date** object using Universal Coordinated Time (UTC).	getUTCDate Method
Returns the day of the week value in a **Date** object using Universal Coordinated Time (UTC).	getUTCDay Method
Returns the year value in a **Date** object using Universal Coordinated Time (UTC).	getUTCFullYear Method
Returns the hours value in a **Date** object using Universal Coordinated Time (UTC).	getUTCHours Method
Returns the milliseconds value in a **Date** object using Universal Coordinated Time (UTC).	getUTCMilliseconds Method
Returns the minutes value in a **Date** object using Universal Coordinated Time (UTC).	getUTCMinutes Method

Returns the month value in a **Date** object using Universal Coordinated Time (UTC).	getUTCMonth Method
Returns the seconds value in a **Date** object using Universal Coordinated Time (UTC).	getUTCSeconds Method
Returns the VT_DATE value in a **Date** object.	getVarDate Method
Returns the year value in a **Date** object.	getYear Method
Returns the character position where the first occurrence of a substring occurs within a **String** object.	indexOf Method
Returns a Boolean value that indicates if a supplied number is finite.	isFinite Method
Returns a Boolean value that indicates whether a value is the reserved value **NaN** (not a number).	isNaN Method
Places HTML <I> tags around text in a **String** object.	italics Method
Returns the current item in the collection.	item Method
Returns a **String** object consisting of all the elements of an array concatenated together.	join Method
Returns the last occurrence of a substring within a **String** object.	lastIndexOf Method
Returns the lowest index value used in the specified dimension of a VBArray.	lbound Method
Places an HTML anchor with an HREF attribute around the text in a **String** object.	link Method
Returns the natural logarithm of a number.	log Method
Returns, as an array, the results of a search on a string using a supplied **Regular Expression** object.	match Method
Returns the greater of two supplied numeric expressions.	max Method
Returns the lesser of two supplied numbers.	min Method
Resets the current item in the collection to the first item.	moveFirst Method
Moves the current item to the next item in the collection.	moveNext Method
Parses a string containing a date, and returns the number of milliseconds between that date and midnight, January 1, 1970.	parse Method
Returns a floating-point number converted from a string.	parseFloat Method
Returns an integer converted from a string.	parseInt Method
Returns the value of a base expression raised to a specified power.	pow Method
Returns a pseudorandom number between 0 and 1.	random Method
Returns a copy of a string with text replaced using a regular expression.	replace Method

Returns an **Array** object with the elements reversed.	reverse Method
Returns a specified numeric expression rounded to the nearest integer.	round Method
Returns the position of the first substring match in a regular expression search.	search Method
Sets the numeric date of the **Date** object using local time.	setDate Method
Sets the year value in the **Date** object using local time.	setFullYear Method
Sets the hour value in the **Date** object using local time.	setHours Method
Sets the milliseconds value in the **Date** object using local time.	setMilliseconds Method
Sets the minutes value in the **Date** object using local time.	setMinutes Method
Sets the month value in the **Date** object using local time.	setMonth Method
Sets the seconds value in the **Date** object using local time.	setSeconds Method
Sets the date and time value in the **Date** object.	setTime Method
Sets the numeric date in the **Date** object using Universal Coordinated Time (UTC).	setUTCDate Method
Sets the year value in the **Date** object using Universal Coordinated Time (UTC).	setUTCFullYear Method
Sets the hours value in the **Date** object using Universal Coordinated Time (UTC).	setUTCHours Method
Sets the milliseconds value in the **Date** object using Universal Coordinated Time (UTC).	setUTCMilliseconds Method
Sets the minutes value in the **Date** object using Universal Coordinated Time (UTC).	setUTCMinutes Method
Sets the month value in the **Date** object using Universal Coordinated Time (UTC).	setUTCMonth Method
Sets the seconds value in the **Date** object using Universal Coordinated Time (UTC).	setUTCSeconds Method
Sets the year value in the **Date** object.	setYear Method
Returns the sine of a number.	sin Method
Returns a section of an array.	slice Method (Array)
Returns a section of a string.	slice Method (String)
Places HTML <SMALL> tags around text in a **String** object.	small Method
Returns an **Array** object with the elements sorted.	sort Method
Returns the array of strings that results when a string is separated into substrings.	split Method

Returns the square root of a number.	sqrt Method
Places HTML <STRIKE> tags around text in a **String** object.	strike Method
Places HTML <SUB> tags around text in a **String** object.	sub Method
Returns a substring beginning at a specified location and having a specified length.	substr Method
Returns the substring at a specified location within a **String** object.	substring Method
Places HTML <SUP> tags around text in a **String** object.	sup Method
Returns the tangent of a number.	tan Method
Returns a Boolean value that indicates whether or not a pattern exists in a searched string.	test Method
Returns a standard JScript array converted from a VBArray.	toArray Method
Returns a date converted to a string using Greenwich Mean Time (GMT).	toGMTString Method
Returns a date converted to a string using the current locale.	toLocaleString Method
Returns a string where all alphabetic characters have been converted to lowercase.	toLowerCase Method
Returns a string representation of an object.	toString Method
Returns a string where all alphabetic characters have been converted to uppercase.	toUpperCase Method
Returns a date converted to a string using Universal Coordinated Time (UTC).	toUTCString Method
Returns the highest index value used in the specified dimension of the VBArray.	ubound Method
Decodes **String** objects encoded with the **escape** method.	unescape Method
Returns the number of milliseconds between midnight, January 1, 1970 Universal Coordinated Time (UTC) (or GMT) and the supplied date.	UTC Method
Returns the primitive value of the specified object.	valueOf Method

---

Build: Topic Version 5.6.9309.1546

JScript

abs Method

Returns the absolute value of a number.

Math.abs(*number***)**

The required *number* argument is a numeric expression for which the absolute value is needed.

Remarks

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

Example

The following example illustrates the use of the **abs** method.

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

Requirements

Version 1

See Also

Math Object Methods

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

acos Method

Returns the arccosine of a number.

Math.acos(*number***)**

The required *number* argument is a numeric expression for which the arccosine is needed.

Remarks

The return value is the arccosine of the *number* argument.

Requirements

Version 1

See Also

asin Method | atan Method | cos Method | sin Method | tan Method

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

anchor Method

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

strVariable.**anchor(***anchorString***)**

Arguments

strVariable
Required. Any **String** object or literal.
anchorString
Required. 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.

Requirements

Version 1

See Also

link Method | String Object Methods | String Object Properties

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

apply Method

Applies a method of an object, substituting another object for the current object.

apply([*thisObj*[,*argArray*]]**)**

Arguments

thisObj
Optional. The object to be used as the current object.
argArray
Optional. Array of arguments to be passed to the function.

Remarks

If *argArray* is not a valid array or is not the **arguments** object, then a TypeError results.

If neither *argArray* nor *thisObj* are supplied, the **global** object is used as *thisObj* and is passed no arguments.

Requirements

Version 5.5

See Also

Applies To: Function Object

Build: Topic Version 5.6.9309.1546

JScript

asin Method

Returns the arcsine of a number.

Math.asin(*number***)**

The required *number* argument is a numeric expression for which the arcsine is needed.

Remarks

The return value is the arcsine of its numeric argument.

Requirements

Version 1

See Also

acos Method | atan Method | cos Method | Math Object Methods | sin Method | tan Method

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

atan Method

Returns the arctangent of a number.

Math.atan(*number*)

The required *number* argument is a numeric expression for which the arctangent is needed.

Remarks

The return value is the arctangent of its numeric argument.

Requirements

Version 1

See Also

acos Method | asin Method | atan2 Method | cos Method | Math Object Methods | sin Method | tan Method

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

atan2 Method

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

```
Math.atan2(y, x)
```

Arguments

x
Required. A numeric expression representing the cartesian x-coordinate.

y
Required. A numeric expression representing the cartesian y-coordinate.

Remarks

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

Requirements

Version 1

See Also

atan Method | tan Method

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

atEnd Method

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

myEnum.**atEnd()**

The required *myEnum* reference is any **Enumerator** object.

Remarks

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**.

Example

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

```
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);
}
```

Requirements

Version 2

See Also

item Method | moveFirst Method | moveNext Method

Applies To: Enumerator Object

Build: Topic Version 5.6.9309.1546

JScript

big Method

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

strVariable.**big( )**

The required *strVariable* reference is any **String** object or literal.

Remarks

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

```
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.

Requirements

Version 1

See Also

small Method | String Object Methods | String Object Properties

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

blink Method

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

```
strVariable.blink( )
```

The required *strVariable* reference is any **String** object or literal.

Remarks

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

```
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.

Requirements

Version 1

See Also

String Object Methods | String Object Properties

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

bold Method

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

```
strVariable.bold()
```

The required *strVariable* reference is any **String** object or literal.

Remarks

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

```
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.

Requirements

Version 1

See Also

italics Method | String Object Methods | String Object Properties

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

call Method

Calls a method of an object, substituting another object for the current object.

```
call([thisObj[, arg1[, arg2[,  [, argN]]]]])
```

Arguments

thisObj
Optional. The object to be used as the current object.
arg1, arg2, , argN
Optional. List of arguments to be passed to the method.

Remarks

The **call** method is used to call a method on behalf of another object. The **call** method allows you to change the object context of a function from the original context to the new object specified by *thisObj*.

If *thisObj* is not supplied, the **global** object is used as *thisObj*.

Requirements

Version 5.5

See Also

Applies To: Function Object

Build: Topic Version 5.6.9309.1546

JScript

ceil Method

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

Math.ceil(*number*)

The required *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.

Requirements

Version 1

See Also

floor Method | Math Object Methods

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

charAt Method

Returns the character at the specified index.

strObj.**charAt(***index***)**

Arguments

strObj
Required. Any **String** object or literal.
index
Required. 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 an empty string.

Example

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

```
function charAtTest(n){
   var str = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"; // Initialize variable.
   var s;                                 // Declare variable.
   s = str.charAt(n - 1);                 // Get correct character
                                          // from position n – 1.
   return(s);                             // Return character.
}
```

Requirements

Version 1

See Also

String Object Methods | String Object Properties

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

charCodeAt Method

Returns an integer representing the Unicode encoding of the character at the specified location.

strObj.**charCodeAt(***index***)**

Arguments

strObj
Required. Any **String** object or literal.

index
Required. Zero-based index of the desired character. Valid values are between 0 and the length of the string minus 1.

Remarks

The first character in a string is at index 0, the second is at index 1, and so forth.

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

Example

The following example illustrates the use of the **charCodeAt** method.

```
function charCodeAtTest(n){
  var str = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"; //Initialize variable.
  var n;                                 //Declare variable.
  n = str.charCodeAt(n - 1);             //Get the Unicode value of the
```

```
                                    // character at position n.
  return(n);                        //Return the value.
}
```

Requirements

Version 5.5

See Also

fromCharCode Methods | String Object methods

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

compile Method

Compiles a regular expression into an internal format for faster execution.

rgExp.**compile(***pattern*, [*flags*]**)**

Arguments

rgExp
Required. An instance of a **Regular Expression** object. Can be a variable name or a literal.
pattern
Required. A string expression containing a regular expression pattern to be compiled
flags

Optional. Available flags, which may be combined, are:

- g (global search for all occurrences of *pattern*)
- i (ignore case)
- m (multiline search)

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. A compiled regular expression speeds things up when reusing the same expression repeatedly. No advantage is gained, however, if the regular expression changes.

Example

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

```
function CompileDemo(){
   var rs;
   var s = "AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPp"
   // Create regular expression for uppercase only.
   var r = new RegExp("[A-Z]", "g");
   var a1 = s.match(r)              // Find matches.
   // Compile the regular expression for lowercase only.
   r.compile("[a-z]", "g");
   var a2 = s.match(r)              // Find matches.
   return(a1 + "\n" + a2;
}
```

Requirements

Version 3

See Also

Regular Expression Object Methods | Regular Expression Object Properties | Regular Expression Syntax

Applies To: Regular Expression Object

Build: Topic Version 5.6.9309.1546

JScript

concat Method (Array)

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

array1.**concat(**[*item1*[, *item2*[, . . . [, *itemN*]]]]**)**

Arguments

array1
Required. The **Array** object to which all other arrays are concatenated.

item1,. . ., *itemN*
Optional. Additional items to add to the end of *array1*.

Remarks

The **concat** method returns an **Array** object containing the concatenation of *array1* and any other supplied items.

The items to be added (*item1* … *itemN*) to the array are added, in order, from left to right. If one of the items is an array, its contents are added to the end of *array1*. If the item is anything other than an array, it is added to the end of the array as a single array element.

Elements of source arrays are copied to the resulting array as follows:

- For an object reference copied from any of the arrays being concatenated to the new array, the object reference continues to point to the same object. A change in either the new array or the original array will result in a change to the other.
- For a numeric or string value being concatenated to the new array, only the value is copied. Changes in a value in one array does not affect the value in the other.

Example

The following example illustrates the use of the **concat** method when used with an array:

```
function ConcatArrayDemo(){
   var a, b, c, d;
   a = new Array(1,2,3);
   b = "JScript";
   c = new Array(42, "VBScript);
   d = a.concat(b, c);
   //Returns the array [1, 2, 3, "JScript", 42, "VBScript"]
   return(d);
}
```

Requirements

Version 3

See Also

concat Method (String) | join Method | String Object

Applies To: Array Object

Build: Topic Version 5.6.9309.1546

JScript

concat Method (String)

Returns a string value containing the concatenation of two or more supplied strings.

string1.**concat(**[*string2*[, *string3*[, . . . [, *stringN*]]]]**)**

Arguments

string1
Required. The **String** object or literal to which all other specified strings are concatenated.
string2,. . ., stringN
Optional. **String** objects or literals to concatenate to the end of *string1*.

Remarks

The result of the **concat** method is equivalent to: *result = string1 + string2 + string3 + … + stringN*. A change of value in either a source or result string does not affect the value in the other string. If any of the arguments are not strings, they are first converted to strings before being concatenated to *string1*.

Example

The following example illustrates the use of the **concat** method when used with a string:

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

Requirements

Version 3

See Also

Addition Operator (+) | Array Object | concat Method (Array) | String Object Methods

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

cos Method

Returns the cosine of a number.

Math.cos(*number***)**

The required *number* argument is a numeric expression for which the cosine is needed.

Remarks

The return value is the cosine of its numeric argument.

Requirements

Version 1

See Also

acos Method | asin Method | atan Method | Math Object Methods | sin Method | tan Method

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

decodeURI Method

Returns the unencoded version of an encoded Uniform Resource Identifier (URI).

decodeURI(*URIstring***)**

The required *URIstring* argument is a value representing an encoded URI.

Remarks

Use the **decodeURI** method instead of the obsolete **unescape** method.

The **decodeURI** method returns a string value.

If the *URIString* is not valid, a URIError occurs.

Requirements

Version 5.5

See Also

decodeURIComponent Method | encodeURI Method

Applies To: Global Object

Build: Topic Version 5.6.9309.1546

JScript

decodeURIComponent Method

Returns the unencoded version of an encoded component of a Uniform Resource Identifier (URI).

decodeURIComponent(*encodedURIString***)**

The required *encodedURIString* argument is a value representing an encoded URI component.

Remarks

A URIComponent is part of a complete URI.

If the *encodedURIString* is not valid, a URIError occurs.

Requirements

Version 5.5

See Also

decodeURI Method | encodeURI Method

Applies To: Global Object

Build: Topic Version 5.6.9309.1546

JScript

dimensions Method

Returns the number of dimensions in a VBArray.

array.**dimensions( )**

The required *array* 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 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(CreateVBArray()));
</SCRIPT>
</BODY>
```

Requirements

Version 3

See Also

getItem Method | lbound Method | toArray Method | ubound Method

Applies To: VBArray Object

Build: Topic Version 5.6.9309.1546

JScript

encodeURI Method

Encodes a text string as a valid Uniform Resource Identifier (URI)

encodeURI(*URIString*)

The required *URIString* argument is a *v*alue representing an encoded URI.

Remarks

The **encodeURI** method returns an encoded URI. If you pass the result to **decodeURI**, the original string is returned. The **encodeURI** method does not encode the following characters: ":", "/", ";", and "?". Use **encodeURIComponent** to encode these characters.

Requirements

Version 5.5

See Also

decodeURI Method | decodeURIComponent Method

Applies To: Global Object

Build: Topic Version 5.6.9309.1546

JScript

encodeURIComponent Method

Encodes a text string as a valid component of a Uniform Resource Identifier (URI).

encodeURIComponent(*encodedURIString*)

The required *encodedURIString* argument is a value representing an encoded URI component.

Remarks

The **encodeURIComponent** method returns an encoded URI. If you pass the result to **decodeURIComponent**, the original string is returned. Because the **encodeURIComponent** method encodes all characters, be careful if the string represents a path such as */folder1/folder2/default.html*. The slash characters will be encoded and will not be valid if sent as a request to a web server. Use the **encodeURI** method if the string contains more than a single URI component.

Requirements

Version 5.5

See Also

decodeURI Method | decodeURIComponent Method

Applies To: Global Object

Build: Topic Version 5.6.9309.1546

JScript

escape Method

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

escape(*charString***)**

The required *charString* argument is any **String** object or literal to be encoded.

Remarks

The **escape** method returns a string value (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 **%u***xxxx* format.

> **Note** The **escape** method should not be used to encode Uniform Resource Identifiers (URI). Use **encodeURI** and **encodeURIComponent** methods instead.

Requirements

Version 1

See Also

encodeURI Method | encodeURIComponent Method | String Object | unescape Method

Applies To: Global Object

Build: Topic Version 5.6.9309.1546

JScript

eval Method

Evaluates JScript code and executes it.

```
eval(codeString)
```

The required *codeString* argument is a string value 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:

```
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.

Requirements

Version 1

See Also

String Object

Applies To: Global Object

Build: Topic Version 5.6.9309.1546

JScript

exec Method

Executes a search on a string using a regular expression pattern, and returns an array containing the results of that search.

```
rgExp.exec(str)
```

Arguments

rgExp
Required. An instance of a **Regular Expression** object containing the regular expression pattern and applicable flags.

str
Required. The **String** object or string literal on which to perform the search.

Remarks

If the **exec** method does not find a match, it returns **null**. If it finds a match, **exec** returns an array, and the properties of the global **RegExp** object are updated to reflect the results of the match. Element zero of the array contains the entire match, while elements 1 – *n* contain any submatches that have occurred within the match. This behavior is identical to the behavior of the **match** method without the global flag (**g**) set.

If the global flag is set for a regular expression, **exec** searches the string beginning at the position indicated by the value of **lastIndex**. If the global flag is not set, **exec** ignores the value of **lastIndex** and searches from the beginning of the string.

The array returned by the **exec** method has three properties, **input**, **index** and **lastIndex.** The **input** property contains the entire searched string. The **index** property contains the position of the matched substring within the complete searched string. The **lastIndex** property contains the position following the last character in the match.

Example

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

```
function RegExpTest(){
  var ver = Number(ScriptEngineMajorVersion() + "." + ScriptEngineMinorVersion())
  if (ver >= 5.5){                  //Test JScript version.
    var src = "The rain in Spain falls mainly in the plain.";
    var re = /\w+/g;                //Create regular expression pattern.
    var arr;
    while ((arr = re.exec(src)) != null)
       document.write(arr.index + "-" + arr.lastIndex + "\t" + arr);
  }
  else{
    alert("You need a newer version of JScript for this to work");
  }
}
```

Requirements

Version 3

See Also

match Method | RegExp Object | Regular Expression Object Methods | Regular Expression Object Properties | Regular Expression Syntax | search method | test Method

Applies To: Regular Expression Object

Build: Topic Version 5.6.9309.1546

JScript

exp Method

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

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

The required *number* argument is numeric expression representing the power of *e*.

Remarks

The return value is e^{number}. The constant *e* is Euler's constant, approximately equal to 2.178 and *number* is the supplied argument.

Requirements

Version 1

See Also

E Property | Math Object Methods

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

fixed Method

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

strVariable.**fixed( )**

The required *strVariable* reference is any String object or literal.

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.

Requirements

Version 1

See Also

String Object Methods | String Object Properties

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

floor Method

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

```
Math.floor(number)
```

The required *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.

Requirements

Version 1

See Also

ceil Method | Math Object Methods

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

fontcolor Method

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

strVariable.**fontcolor(***colorVal***)**

Arguments

strVariable
Required. Any **String** object or literal.

colorVal
Required. String value 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:

```
<FONT COLOR="RED">This is a string</FONT>
```

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.

Requirements

Version 1

See Also

fontsize Method | String Object Methods | String Object Properties

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

fontsize Method

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

strVariable.**fontsize(***intSize***)**

Arguments

strVariable
Required. Any **String** object or literal.
intSize
Required. Integer value that specifies 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.

Requirements

Version 1

See Also

fontcolor Method | String Object Methods | String Object Properties

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

fromCharCode Method

Returns a string from a number of Unicode character values.

```
String.fromCharCode([code1[, code2[, ...[, codeN]]]])
```

Arguments

String
Required. The **String** object.

code1, . . . , codeN
Optional. A series of Unicode character values to convert to a string. If no arguments are supplied, the result is the empty 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, 105, 110);
```

Requirements

Version 3

See Also

charCodeAt Method | String Object Methods

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

getDate Method

Returns the day of the month value in a **Date** object using local time.

dateObj.**getDate()**

The required *dateObj* reference is a **Date** object.

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.

Example

The following example illustrates the use of the **getDate** method.

```
function DateDemo(){
   var d, s = "Today's date is: ";
   d = new Date();
   s += (d.getMonth() + 1) + "/";
   s += d.getDate() + "/";
   s += d.getYear();
   return(s);
}
```

Requirements

Version 1

See Also

Date Object Methods | getUTCDate Method | setDate Method | setUTCDate Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

getDay Method

Returns the day of the week value in a **Date** object using local time.

dateObj.**getDay()**

The required *dateObj* reference is a **Date** object.

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:

Value	Day of the Week
0	Sunday
1	Monday
2	Tuesday
3	Wednesday

4 Thursday
5 Friday
6 Saturday

The following example illustrates the use of the **getDay** method.

```
function DateDemo(){
   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]);
}
```

Requirements

Version 1

See Also

Date Object Methods | getUTCDay Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

getFullYear Method

Returns the year value in the **Date** object using local time.

dateObj.**getFullYear()**

The required *dateObj* reference is a **Date** object.

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 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);
}
```

Requirements

Version 3

See Also

Date Object Methods | getUTCFullYear Method | setFullYear Method | setUTCFullYear Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

getHours Method

Returns the hours value in a **Date** object using local time.

dateObj.**getHours()**

The required *dateObj* reference is a **Date** object.

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.

```
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);
}
```

Requirements

Version 1

See Also

Date Object Methods | getUTCHours Method | setHours Method | setUTCHours Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

getItem Method

Returns the item at the specified location.

safeArray.**getItem(***dimension1*[, *dimension2*, ...], *dimensionN***)**

Arguments

safeArray
Required. A VBArray object.
dimension1, ..., *dimensionN*
Specifies the exact location of the desired element of the VBArray. *n* equals the number of dimensions in the VBArray.

Example

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 CreateVBArray()
   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>
</HEAD>
&ltBODY>
<SCRIPT LANGUAGE="JScript">
<!--
   GetItemTest(CreateVBArray());
-->
</SCRIPT>
</BODY>
```

Requirements

Version 1

See Also

dimensions Method | lbound Method | toArray Method | ubound Method

Applies To: VBArray Object

Build: Topic Version 5.6.9309.1546

JScript

getMilliseconds Method

Returns the milliseconds value in a **Date** object using local time.

```
dateObj.getMilliseconds()
```

The required *dateObj* reference is a **Date** object.

Remarks

To get the number of milliseconds in Universal Coordinated Time (UTC), use the **getUTCMilliseconds** method.

The millisecond value returned can range from 0-999.

Example

The following example illustrates the use of the **getMilliseconds** method.

```
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);
}
```

Requirements

Version 3

See Also

Date Object Methods | getUTCMilliseconds Method | setMilliseconds Method | setUTCMilliseconds Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

getMinutes Method

Returns the minutes value in a **Date** object using local time.

dateObj.**getMinutes()**

The required *dateObj* reference is a **Date** object.

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: when the time is less than one minute after the hour, or 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.

Example

The following example illustrates the use of the **getMinutes** method.

```
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);
}
```

Requirements

Version 3

See Also

Date Object Methods | getUTCMinutes Method | setMinutes Method | setUTCMinutes Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

getMonth Method

Returns the month value in the **Date** object using local time.

dateObj.**getMonth()**

The required *dateObj* reference is a **Date** object.

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.

Example

The following example illustrates the use of the **getMonth** method.

```
function DateDemo(){
   var d, s = "Today's date is: ";
   d = new Date();
   s += (d.getMonth() + 1) + "/";
   s += d.getDate() + "/";
   s += d.getYear();
   return(s);
}
```

Requirements

Version 1

See Also

Date Object Methods | getUTCMonth Method | setMonth Method | setUTCMonth Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

getSeconds Method

Returns the seconds value in a **Date** object using local time.

dateObj.**getSeconds()**

The required *dateObj* reference is a **Date** object.

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.

Example

The following example illustrates the use of the **getSeconds** method.

```
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);
}
```

Requirements

Version 1

See Also

Date Object Methods | getUTCSeconds Method | setSeconds Method | setUTCSeconds Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

getTime Method

Returns the time value in a **Date** object.

dateObj.**getTime()**

The required *dateObj* reference is a **Date** object.

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:

```
var MinMilli = 1000 * 60
var HrMilli = MinMilli * 60
var DyMilli = HrMilli * 24
```

Example

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);
}
```

Requirements

Version 1

See Also

Date Object Methods | setTime Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

getTimezoneOffset Method

Returns the difference in minutes between the time on the host computer and Universal Coordinated Time (UTC).

dateObj.**getTimezoneOffset()**

The required *dateObj* reference is a **Date** object.

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.

Example

The following example illustrates the use of the **getTimezoneOffset** method.

```
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);
}
```

Requirements

Version 1

See Also

Date Object Methods

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

getUTCDate Method

Returns the date in a **Date** object using Universal Coordinated Time (UTC).

dateObj.**getUTCDate()**

The required *dateObj* reference is a **Date** object.

Remarks

To get the date using local time, use the **getDate** method.

The return value is an integer between 1 and 31 that represents the date value in the **Date** object.

Example

The following example illustrates the use of the **getUTCDate** method.

```
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);
}
```

Requirements

Version 3

See Also

Date Object Methods | getDate Method | setDate Method | setUTCDate Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

getUTCDay Method

Returns the day of the week value in a **Date** object using Universal Coordinated Time (UTC).

dateObj.**getUTCDay()**

The required *dateObj* reference is a **Date** object.

Remarks

To get the day of the week using local time, use the **getDate** method.

The value returned by the **getUTCDay** method is an integer between 0 and 6 representing the day of the week and corresponds to a day of the week as follows:

Value	Day of the Week
0	Sunday
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday

Example

The following example illustrates the use of the **getUTCDay** method.

```
function DateDemo(){
   var d, day, x, s = "Today is ";
   var x = new Array("Sunday", "Monday", "Tuesday");
   x = x.concat("Wednesday","Thursday", "Friday");
   x = x.concat("Saturday");
   d = new Date();
   day = d.getUTCDay();
   return(s += x[day]);
}
```

Requirements

Version 3

See Also

Date Object Methods | getDay Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

getUTCFullYear Method

Returns the year value in a **Date** object using Universal Coordinated Time (UTC).

dateObj.**getUTCFullYear()**

The required *dateObj* reference is a **Date** object.

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 year 2000 problem where dates beginning with January 1, 2000 are confused with those beginning with January 1, 1900.

Example

The following example illustrates the use of the **getUTCFullYear** method.

```
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);
}
```

Requirements

Version 3

See Also

Date Object Methods | getFullYear Method | setFullYear Method | setUTCFullYear Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

getUTCHours Method

Returns the hours value in a **Date** object using Universal Coordinated Time (UTC).

dateObj.**getUTCHours()**

The required *dateObj* reference is a **Date** object.

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.

Example

The following example illustrates the use of the **getUTCHours** method.

```
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);
}
```

Requirements

Version 3

See Also

Date Object Methods | getHours Method | setHours Method | setUTCHours Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

getUTCMilliseconds Method

Returns the milliseconds value in a **Date** object using Universal Coordinated Time (UTC).

dateObj.**getUTCMilliseconds()**

The required *dateObj* reference is a **Date** object.

Remarks

To get the number of milliseconds in local time, use the **getMilliseconds** method.

The millisecond value returned can range from 0-999.

Example

The following example illustrates the use of the **getUTCMilliseconds** method.

```
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);
}
```

Requirements

Version 3

See Also

Date Object Methods | getMilliseconds Method | setMilliseconds Method | setUTCMilliseconds Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

getUTCMinutes Method

Returns the minutes value in a **Date** object using Universal Coordinated Time (UTC).

dateObj.**getUTCMinutes()**

The required *dateObj* reference is a **Date** object.

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.

Example

The following example illustrates the use of the **getUTCMinutes** method.

```
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);
}
```

Requirements

Version 3

See Also

Date Object Methods | getMinutes Method | setMinutes Method | setUTCMinutes Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

getUTCMonth Method

Returns the month value in a **Date** object using Universal Coordinated Time (UTC).

dateObj.**getUTCMonth()**

The required *dateObj* reference is a **Date** object.

Remarks

To get the month in local time, use the **getMonth** method.

The **getUTCMonth** 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, **getUTCMonth** returns 0.

Example

The following example illustrates the use of the **getUTCMonth** method.

```
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);
}
```

Requirements

Version 3

See Also

Date Object Methods | getMonth Method | setMonth Method | setUTCMonth Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

getUTCSeconds Method

Returns the seconds value in a **Date** object using Universal Coordinated Time (UTC).

dateObj.**getUTCSeconds()**

The required *dateObj* reference is a **Date** object.

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.

Example

The following example illustrates the use of the **getUTCSeconds** method.

```
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);
}
```

Requirements

Version 3

See Also

Date Object Methods | getSeconds Method | setSeconds Method | setUTCSeconds Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

getVarDate Method

Returns the VT_DATE value in a **Date** object.

dateObj.**getVarDate()**

The required *dateObj* reference is a **Date** object.

Remarks

The **getVarDate** method is used when interacting with COM objects, ActiveX® objects or other objects that accept and return date values in VT_DATE format, such as Visual Basic and VBScript. The actual format is dependent on regional settings and should not be replied upon within JScript.

Requirements

Version 3

See Also

getDate Method | parse Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

getYear Method

Returns the year value in a **Date** object.

dateObj.**getYear()**

The required *dateObj* reference is a **Date** object.

Remarks

This method is obsolete, and is provided for backwards compatibility only. Use the **getFullYear** method instead.

For the years 1900 though 1999, the year is a 2-digit integer value returned as the difference between the stored year and 1900. For dates outside that period, 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 2000 is returned as 100.

Example

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

```
function DateDemo(){
   var d, s = "Today's date is: ";
```

```
   d = new Date();
   s += (d.getMonth() + 1) + "/";
   s += d.getDate() + "/";
   s += d.getYear();
   return(s);
}
```

Requirements

Version 1

See Also

Date Object Methods | getFullYear Method | getUTCFullYear Method | setFullYear Method | setUTCFullYear Method | setYear Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

indexOf Method

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

strObj.**indexOf(***subString*[, *startIndex*]**)**

Arguments

strObj
Required. A **String** object or literal.

subString
Required. Substring to search for within the **String** object.
startIndex
Optional. Integer value specifying the index to begin searching within the **String** object. If omitted, searching starts at the beginning of the string.

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**.

Example

The following example illustrates the use of the **indexOf** method.

```
function IndexDemo(str2){
   var str1 = "BABEBIBOBUBABEBIBOBU"
   var s = str1.indexOf(str2);
   return(s);
}
```

Requirements

Version 1

See Also

lastIndexOf Method | String Object Methods | String Object Properties

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

isFinite Method

Returns a Boolean value that indicates if a supplied number is finite.

isFinite(*number***)**

The required *number* argument is any 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**.

Requirements

Version 3

See Also

isNaN Method

Applies To: Global Object

© 2001 Microsoft Corporation. All rights reserved.

Build: Topic Version 5.6.9309.1546

JScript

isNaN Method

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

isNaN(*numValue***)**

The required *numValue* 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.

Requirements

Version 1

See Also

isFinite Method | NaN Property (Global) | parseFloat Method | parseInt Method

Applies To: Global Object

Build: Topic Version 5.6.9309.1546

JScript

italics Method

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

```
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.

Requirements

Version 1

See Also

bold Method | String Object Methods | String Object Properties

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

item Method

Returns the current item in the collection.

enumObj.**item()**

The required *enumObj* reference is any **Enumerator** object.

Remarks

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

Example

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

```
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);
}
```

Requirements

Version 3

See Also

atEnd Method | moveFirst Method | moveNext Method

Applies To: Enumerator Object

Build: Topic Version 5.6.9309.1546

JScript

join Method

Returns a string value consisting of all the elements of an array concatenated together and separated by the specified separator character.

arrayObj.**join(***separator***)**

Arguments

arrayObj
Required. An **Array** object.
separator
Required. A **String** object used to separate one element of an array from the next in the resulting **String** object. If omitted, the array

elements are separated with a comma.

Remarks

If any element of the array is **undefined** or **null**, it is treated as an empty string.

Example

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

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

Requirements

Version 2

See Also

Array Object Methods | String Object

Applies To: Array Object

Build: Topic Version 5.6.9309.1546

lastIndexOf Method

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

strObj.**lastIndexOf(***substring*[, *startindex*]**)**

Arguments

strObj
Required. A **String** object or literal.
substring
Required. The substring to search for within the **String** object.
startindex
Optional. Integer value specifying the index to begin searching within the **String** object. If omitted, searching begins at the end of the string.

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.

```
function lastIndexDemo(str2)
{
   var str1 = "BABEBIBOBUBABEBIBOBU"
   var s = str1.lastIndexOf(str2);
   return(s);
}
```

Requirements

Version 1

See Also

indexOf Method | String Object Methods | String Object Properties

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

lbound Method

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

safeArray.**lbound(***dimension***)**

Arguments

safeArray
Required. A VBArray object.
dimension
Optional. The dimension of the VBArray for which the lower bound index is wanted. If omitted, **lbound** behaves as if a 1 was passed.

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.

Example

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.

```
<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 lower bound of dimension ";
      s += i + " is ";
      s += a.lbound(i)+ ".<BR>";
      return(s);
   }
}
-->
</SCRIPT>
</HEAD>

<BODY>
```

```
<SCRIPT language="jscript">
   document.write(VBArrayTest(CreateVBArray()));
</SCRIPT>
</BODY>
```

Requirements

Version 3

See Also

dimensions Method | getItem Method | toArray Method | ubound Method

Applies To: VBArray Object

Build: Topic Version 5.6.9309.1546

JScript

link Method

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

```
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:

```
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 a hyperlink</A>
```

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

Requirements

Version 1

See Also

anchor Method | String Object Methods | String Object Properties

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

localeCompare Method

Returns a value indicating whether two strings are equivalent in the current locale.

stringVar.**localeCompare(***stringExp***)**

Arguments

stringVar
Required. A **String** object or literal.
stringExp
Required. String to compare to *stringVar*.

Remarks

The **localeCompare** performs a locale-sensitive string comparison of the *stringVar* and the *stringExp* and returns -1, 0, or +1, depending on the sort order of the system default locale.

If *stringVar* sorts before *stringExp*, **localeCompare** returns –1; if *stringVar* sorts after *stringExp*, +1 is returned. A return value of zero means that the two strings are equivalent.

Requirements

Version 5.5

See Also

toLocalString

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

log Method

Returns the natural logarithm of a number.

Math.log(*number*)

The required *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*.

Requirements

Version 1

See Also

Math Object Methods

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

match Method

Executes a search on a string using a regular expression pattern, and returns an array containing the results of that search.

***stringObj*.match(*rgExp*)**

Arguments

stringObj

Required. The **String** object or string literal on which to perform the search.

rgExp

Required. An instance of a **Regular Expression** object containing the regular expression pattern and applicable flags. Can also be a variable name or string literal containing the regular expression pattern and flags.

Remarks

If the **match** method does not find a match, it returns **null**. If it finds a match, **match** returns an array, and the properties of the global **RegExp** object are updated to reflect the results of the match.

The array returned by the **match** method has three properties, **input**, **index** and **lastIndex.** The **input** property contains the entire searched string. The **index** property contains the position of the matched substring within the complete searched string. The **lastIndex** property contains the position following the last character in the last match.

If the global flag (**g**) is not set, Element zero of the array contains the entire match, while elements 1 – *n* contain any submatches that have occurred within the match. This behavior is identical to the behavior of the **exec** method without the global flag set. If the global flag is set, elements 0 - *n* contain all matches that occurred.

Example

The following example illustrates the use of the **match** method.

```
function MatchDemo(){
   var r, re;         //Declare variables.
   var s = "The rain in Spain falls mainly in the plain";
   re = /ain/i;    //Create regular expression pattern.
   r = s.match(re);   //Attempt match on search string.
   return(r);         //Return first occurrence of "ain".
}
```

This example illustrates the use of the match method with the **g** flag set.

```
function MatchDemo(){
   var r, re;         //Declare variables.
   var s = "The rain in Spain falls mainly in the plain";
   re = /ain/ig;      //Create regular expression pattern.
   r = s.match(re);   //Attempt match on search string.
   return(r);         //Return array containing all four
```

```
                    // occurrences of "ain".
}
```

The following lines of code illustrate the use of a string literal with the **match** method.

```
var r, re = "Spain";
r = "The rain in Spain".replace(re, "Canada");
```

Requirements

Version 3

See Also

exec Method | RegExp Object | replace Method | search Method | String Object Methods | test Method

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

max Method

Returns the greater of zero or more supplied numeric expressions.

Math.max([*number1*[, *number2*[. . . [, *numberN*]]]]**)**

The optional *number1*, *number2*, . . ., *numberN* arguments are numeric expressions to be evaluated.

Remarks

If no arguments are provided, the return value is equal to **NEGATIVE_INFINITY**. If any argument is **NaN**, the return value is also **NaN**.

Requirements

Version 1

See Also

Math Object Methods | min Method| NEGATIVE_INFINITY Property

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

min Method

Returns the lesser of zero or more supplied numeric expressions.

Math.min([*number1*[, *number2*[. . . [,*numberN*]]]]**)**

The optional *number1*, *number2*, . . ., *numberN* arguments are numeric expressions to be evaluated.

Remarks

If no arguments are provided, the return value is equal to **POSITIVE_INFINITY**. If any argument is **NaN**, the return value is also **NaN**.

Requirements

Version 1

See Also

Math Object Methods | max Method | POSITIVE_INFINITY Property

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

moveFirst Method

Resets the current item in the collection to the first item.

enumObj.**moveFirst( )**

The required *enumObj* reference is any **Enumerator** object.

Remarks

If there are no items in the collection, the current item is set to undefined.

Example

In following example, the **moveFirst** method is used to evaluate members of the **Drives** collection from the beginning of the list:

```
function ShowFirstAvailableDrive(){
   var fso, s, e, x;                //Declare variables.
   fso = new ActiveXObject("Scripting.FileSystemObject");
```

```
   e = new Enumerator(fso.Drives);  //Create Enumerator object.
   e.moveFirst();                   //Move to first drive.
   s = "";                          //Initialize s.
   do
   {
      x = e.item();                 //Test for existence of drive.
      if (x.IsReady)                //See if it's ready.
      {
         s = x.DriveLetter + ":";   //Assign 1st drive letter to s.
         break;
      }
      else
         if (e.atEnd())             //See if at the end of the collection.
         {
            s = "No drives are available";
            break;
         }
      e.moveNext();                 //Move to the next drive.
   }
   while (!e.atEnd());              //Do while not at collection end.
   return(s);                       //Return list of available drives.
}
```

Requirements

Version 3

See Also

atEnd Method | item Method | moveNext Method

Applies To: Enumerator Object

Build: Topic Version 5.6.9309.1546

JScript

moveNext Method

Moves the current item to the next item in the collection.

enumObj.**moveNext( )**

The required *enumObj* reference 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:

```
function ShowDriveList(){
   var fso, s, n, e, x;                 //Declare variables.
   fso = new ActiveXObject("Scripting.FileSystemObject");
   e = new Enumerator(fso.Drives);      //Create Enumerator object.
   s = "";                              //Initialize s.
   for (; !e.atEnd(); e.moveNext())
   {
      x = e.item();
      s = s + x.DriveLetter;            //Add drive letter
      s += " - ";                       //Add "-" character.
      if (x.DriveType == 3)
         n = x.ShareName;               //Add share name.
      else if (x.IsReady)
         n = x.VolumeName;              //Add volume name.
      else
         n = "[Drive not ready]";       //Indicate drive not ready.
      s +=   n + "\n";
   }
   return(s);                           //Return drive status.
}
```

Requirements

Version 3

See Also

atEnd Method | item Method | moveFirst Method

Applies To: Enumerator Object

Build: Topic Version 5.6.9309.1546

JScript

parse Method

Parses a string containing a date, and returns the number of milliseconds between that date and midnight, January 1, 1970.

Date.parse(*dateVal***)**

The required *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.

```
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.

Example

The following example illustrates the use of the **parse** method. Provide the function with a date and the function will return the difference between the date provided and 1/1/1970:

```
function GetTimeTest(testdate){
   var s, t;                    //Declare variables.
   var MinMilli = 1000 * 60;       //Initialize variables.
   var HrMilli = MinMilli * 60;
   var DyMilli = HrMilli * 24;
   t = Date.parse(testdate);       //Parse testdate.
   s = "There are "                //Create return string.
   s += Math.round(Math.abs(t / DyMilli)) + " days "
   s += "between " + testdate + " and 1/1/70";
   return(s);                      //Return results.
}
```

Requirements

Version 1

See Also

Date Object Methods

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

parseFloat Method

Returns a floating-point number converted from a string.

parseFloat(*numString***)**

The required *numString* argument is a string that contains a floating-point number.

Remarks

The **parseFloat** method returns a 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.

```
parseFloat("abc")      // Returns NaN.
parseFloat("1.2abc")   // Returns 1.2.
```

You can test for **NaN** using the **isNaN** method.

Requirements

Version 1

See Also

isNaN Method | parseInt Method | String Object

Applies To: Global Object

Build: Topic Version 5.6.9309.1546

JScript

parseInt Method

Returns an integer converted from a string.

```
parseInt(numString, [radix])
```

Arguments

numString
Required. A string to convert into a number.

radix
Optional. A value between 2 and 36 indicating the base of the number contained in *numString*. 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.

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.

```
parseInt("abc")     // Returns NaN.
parseInt("12abc")   // Returns 12.
```

You can test for **NaN** using the **isNaN** method.

Requirements

Version 1

See Also

isNaN Method | parseFloat Method | String Object | valueOf Method

Applies To: Global Object

Build: Topic Version 5.6.9309.1546

JScript

pop Method

Removes the last element from an array and returns it.

```
arrayObj.pop( )
```

The required *arrayObj* reference is an **Array** object.

Remarks

If the array is empty, **undefined** is returned.

Requirements

Version 5.5

See Also

push Method

Applies To: Array Object

Build: Topic Version 5.6.9309.1546

JScript

pow Method

Returns the value of a base expression taken to a specified power.

```
Math.pow(base, exponent)
```

Arguments

base
Required. The base value of the expression.
exponent
Required. The exponent value of the expression.

Example

In the following example, a numeric expression equal to base^{exponent} returns 1000.

```
Math.pow(10,3);
```

Requirements

Version 1

See Also

Math Object Methods

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

push Method

Appends new elements to an array, and returns the new length of the array.

```
arrayObj.push([item1 [item2 [. . . [itemN ]]]])
```

Arguments

arrayObj
Required. An **Array** object.
item, item2,. . ., itemN
Optional. New elements of the **Array**.

Remarks

The **push** method appends elements in the order in which they appear. If one of the arguments is an array, it is added as a single element. Use the **concat** method to join the elements from two or more arrays.

Requirements

Version 5.5

See Also

concat Method | pop Method

Applies To: Array Object

Build: Topic Version 5.6.9309.1546

JScript

random Method

Returns a pseudorandom number between 0 and 1.

```
Math.random( )
```

Remarks

The pseudorandom number generated is from 0 (inclusive) to 1 (exclusive), that is, the returned number can be zero, but it will always be less than one. The random number generator is seeded automatically when JScript is first loaded.

Requirements

Version 1

See Also

Math Object Methods

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

replace Method

Returns a copy of a string with text replaced using a regular expression or search string.

```
stringObj.replace(rgExp, replaceText)
```

Arguments

stringObj
Required. The **String** object or string literal on which to perform the replacement. This string is not modified by the **replace** method.

rgExp
Required. An instance of a **Regular Expression** object containing the regular expression pattern and applicable flags. Can also be a **String** object or literal. If *rgExp* is not an instance of a **Regular Expression** object, it is converted to a string, and an exact search is made for the results; no attempt is made to convert the string into a regular expression.

replaceText
Required. A **String** object or string literal containing the text to replace for every successful match of *rgExp* in *stringObj*. In JScript 5.5 or later, the *replaceText* argument can also be a function that returns the replacement text.

Remarks

The result of the **replace** method is a copy of *stringObj* after the specified replacements have been made.

Any of the following match variables can be used to identify the most recent match and the string from which it came. The match variables

can be used in text replacement where the replacement string has to be determined dynamically.

Characters	Meaning
$$	**$** (JScript 5.5 or later)
$&	Specifies that portion of *stringObj* that the entire pattern matched. (JScript 5.5 or later)
$`	Specifies that portion of *stringObj* that precedes the match described by **$&**. (JScript 5.5 or later)
$'	Specifies that portion of *stringObj* that follows the match described by **$&**. (JScript 5.5 or later)
$*n*	The *n*th captured submatch, where *n* is a single decimal digit from 1 through 9. (JScript 5.5 or later)
$*nn*	The *nn*th captured submatch, where *nn* is a two-digit decimal number from 01 through 99. (JScript 5.5 or later)

If *replaceText* is a function, for each matched substring the function is called with the following *m* + 3 arguments where *m* is the number of left capturing parentheses in the *rgExp*. The first argument is the substring that matched. The next *m* arguments are all of the captures that resulted from the search. Argument *m* + 2 is the offset within *stringObj* where the match occurred, and argument *m* + 3 is *stringObj*. The result is the string value that results from replacing each matched substring with the corresponding return value of the function call.

The **replace** method updates the properties of the global **RegExp** object.

Example

The following example illustrates the use of the **replace** method to replace the first instance of the word "The" with the word "A."

```
function ReplaceDemo(){
   var r, re;                    //Declare variables.
   var ss = "The man hit the ball with the bat.\n";
   ss += "while the fielder caught the ball with the glove.";
   re = /The/g;              //Create regular expression pattern.
   r = ss.replace(re, "A");    //Replace "A" with "The".
   return(r);                   //Return string with replacement made.
}
```

In addition, the **replace** method can also replace subexpressions in the pattern. The following example swaps each pair of words in the string.

```
function ReplaceDemo(){
   var r, re;                      //Declare variables.
   var ss = "The rain in Spain falls mainly in the plain.";
   re = /(\S+)(\s+)(\S+)/g;        //Create regular expression pattern.
   r = ss.replace(re, "$3$2$1");   //Swap each pair of words.
   return(r);                      //Return resulting string.
```

```
}
```

The following example, which works in JScript 5.5 and later, performs a Fahrenheit to Celsius conversion, illustrates using a function as *replaceText*. To see how this function works, pass in a string containing a number followed immediately by an "F" (e.g., "Water boils at 212").

```
function f2c(s) {
  var test = /(\d+(\.\d*)?)F\b/g;    //Initialize pattern.
  return(s.replace
    (test,
      function($0,$1,$2) {
        return((($1-32) * 5/9) + "C");
      }
    )
  );
}
document.write(f2c("Water freezes at 32F and boils at 212F."));
```

Requirements

Version 1

See Also

exec Method | match Method | RegExp Object | search Method | String Object Methods | test Method

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

reverse Method

Returns an **Array** object with the elements reversed.

arrayObj.**reverse( )**

The required *arrayObj* reference is an **Array** object.

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.

Example

The following example illustrates the use of the **reverse** method.

```
function ReverseDemo(){
   var a, l;                  //Declare variables.
   a = new Array(0,1,2,3,4);  //Create an array and populate it.
   l = a.reverse();           //Reverse the contents of the array.
   return(l);                 //Return the resulting array.
}
```

Requirements

Version 2

See Also

Array Object Methods

Applies To: Array Object

Build: Topic Version 5.6.9309.1546

JScript

round Method

Returns a supplied numeric expression rounded to the nearest integer.

Math.round(*number***)**

The required *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*.

Requirements

Version 1

See Also

Math Object Methods

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

search Method

Returns the position of the first substring match in a regular expression search.

stringObj.**search(***rgExp***)**

Arguments

stringObj
Required. The **String** object or string literal on which to perform the search.

rgExp
Required. An instance of a **Regular Expression** object containing the regular expression pattern and applicable flags.

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.

Example

The following example illustrates the use of the **search** method.

```
function SearchDemo(){
   var r, re;                  //Declare variables.
   var s = "The rain in Spain falls mainly in the plain.";
   re = /falls/i;            //Create regular expression pattern.
   r = s.search(re);           //Search the string.
   return(r);                  //Return the Boolean result.
}
```

Requirements

Version 3

See Also

exec Method | match Method | Regular Expression Object | Regular Expression Syntax | replace Method | String Object Methods | test Method

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

setDate Method

Sets the numeric date of the **Date** object using local time.

```
dateObj.setDate(numDate)
```

Arguments

dateObj
Required. Any **Date** object.

numDate
Required. 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.

Example

The following example illustrates the use of the **setDate** method.

```
function SetDateDemo(newdate){
   var d, s;                  //Declare variables.
   d = new Date();            //Create date object.
   d.setDate(newdate);        //Set date to newdate.
   s = "Current setting is ";
   s += d.toLocaleString();
   return(s);                 //Return newly set date.
}
```

Requirements

Version 3

See Also

Date Object Methods | getDate Method | getUTCDate Method | setUTCDate Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

setFullYear Method

Sets the year value in the **Date** object using local time.

dateObj.**setFullYear(***numYear*[, *numMonth*[, *numDate*]]**)**

Arguments

dateObj
Required. Any **Date** object.
numYear
Required. A numeric value equal to the year.
numMonth
Optional. A numeric value equal to the month. Must be supplied if *numDate* is supplied.
numDate
Optional. A numeric value equal to the date.

Remarks

All **set** methods taking optional arguments use the value returned from corresponding **get** methods, if you do not specify the 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.

Example

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

```
function SetFullYearDemo(newyear){
   var d, s;                  //Declare variables.
   d = new Date();            //Create Date object.
   d.setFullYear(newyear);    //Set year.
   s = "Current setting is ";
   s += d.toLocaleString();
   return(s);                 //Return new date setting.
}
```

Requirements

Version 3

See Also

Date Object Methods | getFullYear Method | getUTCFullYear Method | setUTCFullYear Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

setHours Method

Sets the hour value in the **Date** object using local time.

```
dateObj.setHours(numHours[, numMin[, numSec[, numMilli]]])
```

Arguments

dateObj
Required. Any **Date** object.
numHours
Required. A numeric value equal to the hours value.
numMin
Optional. A numeric value equal to the minutes value. Must be supplied if either of the following arguments is used.
numSec
Optional. A numeric value equal to the seconds value. Must be supplied if the following argument is used.
numMilli

Optional. A numeric value equal to the milliseconds value.

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 *numMinutes* argument is optional, but not specified, JScript uses the value returned from the **getMinutes** 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.

Example

The following example illustrates the use of the **setHours** method.

```
function SetHoursDemo(nhr, nmin, nsec){
   var d, s;                     //Declare variables.
   d = new Date();               //Create Date object.
   d.setHours(nhr, nmin, nsec);  //Set hours, minutes, & seconds.
   s = "Current setting is " + d.toLocaleString()
   return(s);                    //Return new date setting.
}
```

Requirements

Version 3

See Also

Date Object Methods | getHours Method | getUTCHours Method | setUTCHours Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

setMilliseconds Method

Sets the milliseconds value in the **Date** object using local time.

dateObj.**setMilliseconds(***numMilli***)**

Arguments

dateObj
Required. Any **Date** object.
numMilli
Required. A numeric value equal to the millisecond value.

Remarks

To set the milliseconds value using Universal Coordinated Time (UTC), use the **setUTCMilliseconds** method.

If the value of *numMilli* is greater than 999 or is a negative number, the stored number of seconds (and minutes, hours, and so forth if necessary) is incremented an appropriate amount.

Example

The following example illustrates the use of the **setMilliseconds** method.

```
function SetMSecDemo(nmsec){
   var d, s;                    //Declare variables.
   var sep = ":";               //Initialize separator.
   d = new Date();              //Create Date object.
   d.setMilliseconds(nmsec);    //Set milliseconds.
   s = "Current setting is ";
   s += d.toLocaleString() + sep + d.getMilliseconds();
```

```
   return(s);                  //Return new date setting.
}
```

Requirements

Version 3

See Also

Date Object Methods | getMilliseconds Method | getUTCMilliseconds Method | setUTCMilliseconds Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

setMinutes Method

Sets the minutes value in the **Date** object using local time.

dateObj.**setMinutes(***numMinutes*[, *numSeconds*[, *numMilli*]]**)**

Arguments

dateObj
Required. Any **Date** object.
numMinutes
Required. A numeric value equal to the minutes value.
numSeconds
Optional. A numeric value equal to the seconds value. Must be supplied if the *numMilli* argument is used.

numMilli
Optional. A numeric value equal to the milliseconds value.

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 *numSeconds* argument is optional, but not specified, JScript uses the value returned from the **getSeconds** 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.

Example

The following example illustrates the use of the **setMinutes** method.

```
function SetMinutesDemo(nmin, nsec){
   var d, s;                     //Declare variables.
   d = new Date();               //Create Date object.
   d.setMinutes(nmin, nsec);     //Set minutes.
   s = "Current setting is " + d.toLocaleString()
   return(s);                    //Return new setting.
}
```

Requirements

Version 1

See Also

Date Object Methods | getMinutes Method | getUTCMinutes Method | setUTCMinutes Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

setMonth Method

Sets the month value in the **Date** object using local time.

dateObj.**setMonth(***numMonth*[, *dateVal*]**)**

Arguments

dateObj
Required. Any **Date** object.
numMonth
Required. A numeric value equal to the month.
dateVal
Optional. A numeric value representing the date. If not supplied, the value from a call to the **getDate** method is used.

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."

Example

The following example illustrates the use of the **setMonth** method.

```
function SetMonthDemo(newmonth){
   var d, s;              //Declare variables.
```

```
   d = new Date();         //Create Date object.
   d.setMonth(newmonth);   //Set month.
   s = "Current setting is ";
   s += d.toLocaleString();
   return(s);              //Return new setting.
}
```

Requirements

Version 1

See Also

Date Object Methods | getMonth Method | getUTCMonth Method | setUTCMonth Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

setSeconds Method

Sets the seconds value in the **Date** object using local time.

dateObj.**setSeconds(***numSeconds*[, *numMilli*]**)**

Arguments

dateObj
Required. Any **Date** object.

numSeconds
Required. A numeric value equal to the seconds value.
numMilli
Optional. A numeric value equal to the milliseconds value.

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 *numMilli* argument is optional, but not specified, JScript uses the value returned from the **getMilliseconds** 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."

Example

The following example illustrates the use of the **setSeconds** method.

```
function SetSecondsDemo(nsec, nmsec){
   var d, s;                  //Declare variables.
   var sep = ":";
   d = new Date();            //Create Date object.
   d.setSeconds(nsec, nmsec);  //Set seconds and milliseconds.
   s = "Current setting is ";
   s += d.toLocaleString() + sep + d.getMilliseconds();
   return(s);                 //Return new setting.
}
```

Requirements

Version 1

See Also

Date Object Methods | getSeconds Method | getUTCSeconds Method | setUTCSeconds Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

setTime Method

Sets the date and time value in the **Date** object.

dateObj.**setTime(***milliseconds***)**

Arguments

dateObj
Required. Any **Date** object.
milliseconds
Required. 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.

Example

The following example illustrates the use of the **setTime** method.

```
function SetTimeTest(newtime){
```

```
   var d, s;                  //Declare variables.
   d = new Date();            //Create Date object.
   d.setTime(newtime);        //Set time.
   s = "Current setting is ";
   s += d.toUTCString();
   return(s);                 //Return new setting.
}
```

Requirements

Version 1

See Also

Date Object Methods | getTime Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

setUTCDate Method

Sets the numeric date in the **Date** object using Universal Coordinated Time (UTC).

dateObj.**setUTCDate(***numDate***)**

Arguments

dateObj
Required. Any **Date** object.

numDate
Required. A numeric value equal to the numeric date.

Remarks

To set the date using local time, use the **setDate** 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 **setUTCDate (32)** is called, the date changes to February 1, 1996. Negative numbers have a similar behavior.

Example

The following example illustrates the use of the **setUTCDate** method.

```
function SetUTCDateDemo(newdate){
   var d, s;                    //Declare variables.
   d = new Date();              //Create Date object.
   d.setUTCDate(newdate);       //Set UTC date.
   s = "Current setting is ";
   s += d.toUTCString();
   return(s);                   //Return new setting.
}
```

Requirements

Version 3

See Also

Date Object Methods | getDate Method | getUTCDate Method | setDate Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

setUTCFullYear Method

Sets the year value in the **Date** object using Universal Coordinated Time (UTC).

```
dateObj.setUTCFullYear(numYear[, numMonth[, numDate]])
```

Arguments

dateObj
Required. Any **Date** object.
numYear
Required. A numeric value equal to the year.
numMonth
Optional. A numeric value equal to the month. Must be supplied if *numDate* is supplied.
numDate
Optional. A numeric value equal to the date.

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 **getUTCMonth** 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.

Example

The following example illustrates the use of the **setUTCFullYear** method.

```
function SetUTCFullYearDemo(newyear){
   var d, s;                   //Declare variables.
   d = new Date();             //Create Date object.
   d.setUTCFullYear(newyear);  //Set UTC full year.
   s = "Current setting is ";
   s += d.toUTCString();
   return(s);                  //Return new setting.
}
```

Requirements

Version 3

See Also

Date Object Methods | getFullYear Method | getUTCFullYear Method | setFullYear Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

setUTCHours Method

Sets the hours value in the **Date** object using Universal Coordinated Time (UTC).

dateObj.**setUTCHours(***numHours*[, *numMin*[, *numSec*[, *numMilli*]]]**)**

Arguments

dateObj

Required. Any **Date** object.

numHours

Required. A numeric value equal to the hours value.

numMin

Optional. A numeric value equal to the minutes value. Must be supplied if either *numSec* or *numMilli* are used.

numSec

Optional. A numeric value equal to the seconds value. Must be supplied if *numMilli* argument is used.

numMilli

Optional. A numeric value equal to the milliseconds value.

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 *numMin* argument is optional, but not specified, JScript uses the value returned from the **getUTCMinutes** 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."

Example

The following example illustrates the use of the **setUTCHours** method.

```
function SetUTCHoursDemo(nhr, nmin, nsec){
   var d, s;                      //Declare variables.
   d = new Date();                //Create Date object.
   d.setUTCHours(nhr, nmin, nsec);  //Set UTC hours, minutes, seconds.
   s = "Current setting is " + d.toUTCString()
   return(s);                     //Return new setting.
}
```

Requirements

Version 3

See Also

Date Object Methods | getHours Method | getUTCHours Method | setHours Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

setUTCMilliseconds Method

Sets the milliseconds value in the **Date** object using Universal Coordinated Time (UTC).

```
dateObj.setUTCMilliseconds(numMilli)
```

Arguments

dateObj
Required. Any **Date** object.

numMilli
Required. A numeric value equal to the millisecond value.

Remarks

To set the milliseconds using local time, use the **setMilliseconds** method.

If the value of *numMilli* is greater than 999, or is a negative number, the stored number of seconds (and minutes, hours, and so forth, if necessary) is incremented an appropriate amount.

Example

The following example illustrates the use of the **setUTCMilliseconds** method.

```
function SetUTCMSecDemo(nmsec){
   var d, s;                          //Declare variables.
   var sep = ":";                     //Initialize separator.
   d = new Date();                    //Create Date object.
   d.setUTCMilliseconds(nmsec);       //Set UTC milliseconds.
   s = "Current setting is ";
   s += d.toUTCString() + sep + d.getUTCMilliseconds();
   return(s);                         //Return new setting.
}
```

Requirements

Version 3

See Also

Date Object Methods | getMilliseconds Method | getUTCMilliseconds Method | setMilliseconds Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

setUTCMinutes Method

Sets the minutes value in the **Date** object using Universal Coordinated Time (UTC).

dateObj.**setUTCMinutes(***numMinutes*[, *numSeconds*[, *numMilli*]]**)**

Arguments

dateObj
Required. Any **Date** object.
numMinutes
Required. A numeric value equal to the minutes value.
numSeconds
Optional. A numeric value equal to the seconds value. Must be supplied if *numMilli* is used.
numMilli
Optional. A numeric value equal to the milliseconds value.

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 *numSeconds* argument is optional, but not specified, JScript uses the value returned from the **getUTCSeconds** 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."

Example

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

```
function SetUTCMinutesDemo(nmin, nsec){
   var d, s;                    //Declare variables.
   d = new Date();              //Create Date object.
   d.setUTCMinutes(nmin,nsec);  //Set UTC minutes.
   s = "Current setting is " + d.toUTCString()
   return(s);                   //Return new setting.
}
```

Requirements

Version 3

See Also

Date Object Methods | getMinutes Method | getUTCMinutes Method | setMinutes Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

setUTCMonth Method

Sets the month value in the **Date** object using Universal Coordinated Time (UTC).

dateObj.**setUTCMonth(***numMonth*[, *dateVal*]**)**

Arguments

dateObj
Required. Any **Date** object.
numMonth
Required. A numeric value equal to the month.
dateVal
Optional. A numeric value representing the date. If not supplied, the value from a call to the **getUTCDate** method is used.

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 **setUTCMonth(14)** is called, the date is changed to "Mar 5,

1997 00:00:00.00."

Example

The following example illustrates the use of the **setUTCMonth** method.

```
function SetUTCMonthDemo(newmonth){
   var d, s;                        //Declare variables.
   d = new Date();                  //Create Date object.
   d.setUTCMonth(newmonth);         //Set UTC month.
   s = "Current setting is ";
   s += d.toUTCString();
   return(s);                       //Return new setting.
}
```

Requirements

Version 3

See Also

Date Object Methods | getMonth Method | getUTCMonth Method | setMonth Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

setUTCSeconds Method

Sets the seconds value in the **Date** object using Universal Coordinated Time (UTC).

```
dateObj.setUTCSeconds(numSeconds[, numMilli])
```

Arguments

dateObj
Required. Any **Date** object.
numSeconds
Required. A numeric value equal to the seconds value.
numMilli
Optional. A numeric value equal to the milliseconds value.

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 *numMilli* argument is optional, but not specified, JScript uses the value returned from the **getUTCMilliseconds** 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."

Example

The following example illustrates the use of the **setSeconds** method.

```
function SetUTCSecondsDemo(nsec, nmsec){
   var d, s;                        //Declare variables.
   d = new Date();                  //Create Date object.
   d.setUTCSeconds(nsec, nmsec);    //Set UTC seconds and milliseconds.
   s = "Current UTC milliseconds setting is ";
   s += d.getUTCMilliseconds();     //Get new setting.
   return(s);                       //Return new setting.
}
```

Requirements

Version 3

See Also

Date Object Methods | getSeconds Method | getUTCSeconds Method | setSeconds Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

setYear Method

Sets the year value in the **Date** object.

dateObj.**setYear(***numYear***)**

Arguments

dateObj
Required. Any **Date** object.
numYear
Required. 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 *numYear*, regardless of the value of the year. For example, to set the year to 1899 *numYear* is -1 and to set the year 2000 *numYear* is 100.

Requirements

Version 1

See Also

Date Object Methods | getFullYear Method | getUTCFullYear Method | getYear Method | setFullYear Method | setUTCFullYear Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

shift Method

Removes the first element from an array and returns it.

arrayObj.**shift( )**

The required *arrayObj* reference is an **Array** object.

Remarks

The **shift** method removes the first element from an array and returns it.

Requirements

Version 5.5

See Also

unshift Method

Applies To: Array Object

Build: Topic Version 5.6.9309.1546

JScript

sin Method

Returns the sine of a number.

```
Math.sin(number)
```

The *number* argument is a numeric expression for which the sine is needed.

Remarks

The return value is the sine of the numeric argument.

Requirements

Version 1

See Also

acos Method | asin Method | atan Method | cos Method | Math Object Methods | tan Method

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

slice Method (Array)

Returns a section of an array.

arrayObj.**slice(***start*, [*end*]**)**

Arguments

arrayObj
Required. An **Array** object.

start
Required. The index to the beginning of the specified portion of *arrayObj*.

end
Optional. The index to the end of the specified portion of *arrayObj*.

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 *start* is negative, it is treated as *length* + *start* where *length* is the length of the array. If *end* is negative, it is treated as *length* + *end* where *length* is the length of the array. If *end* is omitted, extraction

continues to the end of *arrayObj*. If *end* occurs before *start*, no elements are copied to the new array.

Example

In the following example, all but the last element of *myArray* is copied into *newArray*:

```
newArray = myArray.slice(0, -1)
```

Requirements

Version 3

See Also

slice Method (String) | String Object

Applies To: Array Object

Build: Topic Version 5.6.9309.1546

JScript

slice Method (String)

Returns a section of a string.

```
stringObj.slice(start, [end])
```

Arguments

stringObj
Required. A **String** object or literal.
start
Required. The index to the beginning of the specified portion of *stringObj*.
end
Optional. The index to the end of the specified portion of *stringObj*.

Remarks

The **slice** method returns a **String** object containing the specified portion of *stringObj*.

The **slice** method copies up to, but not including, the element indicated by *end*. If *start* is negative, it is treated as *length* + *start* where *length* is the length of the string. If *end* is negative, it is treated as *length* + *end* where *length* is the length of the string. If *end* is omitted, extraction continues to the end of *stringObj*. If *end* occurs before *start*, no characters are copied to the new string.

Example

In the following example, the two uses of the **slice** method return the same result. In the second example, negative one (-1) points to the last character in `str1` as the ending point.

```
str1.slice(0)
str2.slice(0,-1)
```

Requirements

Version 3

See Also

Array Object | slice Method (Array) | String Object Methods

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

small Method

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

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

Remarks

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

```
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.

Requirements

Version 1

See Also

big Method | String Object Methods | String Object Properties

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

sort Method

Returns an **Array** object with the elements sorted.

arrayobj.**sort(***sortFunction***)**

Arguments

arrayObj
Required. Any **Array** object.

sortFunction
Optional. 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.

Example

The following example illustrates the use of the **sort** method.

```
function SortDemo(){
   var a, l;                      //Declare variables.
```

```
   a = new Array("X" ,"y" ,"d", "Z", "v","m","r");
   l = a.sort();                    //Sort the array.
   return(l);                       //Return sorted array.
}
```

Requirements

Version 2

See Also

Array Object Methods

Applies To: Array Object

Build: Topic Version 5.6.9309.1546

JScript

splice Method

Removes elements from an array and, if necessary, inserts new elements in their place, returning the deleted elements.

arrayObj.**splice(***start, deleteCount,* [*item1*[, *item2*[, . . . [,*itemN*]]]]**)**

Arguments

arrayObj
Required. An **Array** object.

start
Required. The zero-based location in the array from which to start removing elements.

deleteCount
Required. The number of elements to remove.
item1, item2,. . ., itemN
Optional. Elements to insert into the array in place of the deleted elements.

Remarks

The **splice** method modifies *arrayObj* by removing the specified number of elements from position *start* and inserting new elements. The deleted elements are returned as a new **array** object.

Requirements

Version 5.5

See Also

slice Method (Array)

Applies To: Array Object

Build: Topic Version 5.6.9309.1546

JScript

split Method

Returns the array of strings that results when a string is separated into substrings.

```
stringObj.split([separator[, limit]])
```

Arguments

stringObj
Required. The **String** object or literal to be split. This object is not modified by the **split** method.

separator
Optional. A string or an instance of a **Regular Expression** object identifying one or more characters to use in separating the string. If omitted, a single-element array containing the entire string is returned.

limit
Optional. A value used to limit the number of elements returned in the array.

Remarks

The result of the **split** method is an array of strings split at each point where *separator* occurs in *stringObj*. The *separator* is not returned as part of any array element.

Example

The following example illustrates the use of the **split** method.

```
function SplitDemo(){
   var s, ss;
   var s = "The rain in Spain falls mainly in the plain.";
   // Split at each space character.
   ss = s.split(" ");
   return(ss);
}
```

Requirements

Version 3

See Also

concat Method | RegExp Object | Regular Expression Object | Regular Expression Syntax | String Object Methods

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

sqrt Method

Returns the square root of a number.

Math.sqrt(*number***)**

The required *number* argument is a numeric expression.

Remarks

If *number* is negative, the return value is **NaN**.

Requirements

Version 1

See Also

Math Object Methods | SQRT1_2 Property | SQRT2 Property

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

strike Method

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

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

Remarks

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

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

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

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

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

Requirements

Version 1

See Also

String Object Methods | String Object Properties

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

sub Method

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

```
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.

Requirements

Version 1

See Also

String Object Methods | String Object Properties | sup Method

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

substr Method

Returns a substring beginning at a specified location and having a specified length.

stringvar.**substr(***start* [, *length*]**)**

Arguments

stringvar
Required. A string literal or **String** object from which the substring is extracted.

start
Required. The starting position of the desired substring. The index of the first character in the string is zero.

length
Optional. The number of characters to include in the returned substring.

Remarks

If *length* is zero or negative, an empty string is returned. If not specified, the substring continues to the end of *stringvar*.

Example

The following example illustrates the use of the **substr** method.

```
function SubstrDemo(){
   var s, ss;                //Declare variables.
   var s = "The rain in Spain falls mainly in the plain.";
   ss = s.substr(12, 5);  //Get substring.
```

```
   return(ss);               // Returns "Spain".
}
```

Requirements

Version 3

See Also

String Object Methods | String Object Properties | substring Method

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

substring Method

Returns the substring at the specified location within a **String** object.

```
strVariable.substring(start, end)
"String Literal".substring(start, end)
```

Arguments

start
The zero-based index integer indicating the beginning of the substring.

end
The zero-based index integer indicating the end of the substring.

Remarks

The **substring** method returns a string containing the substring from *start* up to, but not including, *end*.

The **substring** method uses the lower value 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.

If either *start* or *end* is **NaN** or negative, it is replaced with zero.

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.

Example

The following example illustrates the use of the **substring** method.

```
function SubstringDemo(){
   var ss;                         //Declare variables.
   var s = "The rain in Spain falls mainly in the plain..";
   ss = s.substring(12, 17);   //Get substring.
   return(ss);                     //Return substring.
}
```

Requirements

Version 1

See Also

String Object Methods | String Object Properties | substr Method

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

sup Method

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

```
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.

Requirements

Version 1

See Also

String Object Methods | String Object Properties | sub Method

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

tan Method

Returns the tangent of a number.

Math.tan(*number*)

The required *number* argument is a numeric expression for which the tangent is sought.

Remarks

The return value is the tangent of *number*.

Requirements

Version 1

See Also

acos Method | asin Method | atan Method | atan2 Method | cos Method | Math Object Methods | sin Method

Applies To: Math Object

© 2001 Microsoft Corporation. All rights reserved.

Build: Topic Version 5.6.9309.1546

JScript

test Method

Returns a Boolean value that indicates whether or not a pattern exists in a searched string.

rgExp.**test(***str***)**

Arguments

rgExp
Required. An instance of a **Regular Expression** object containing the regular expression pattern and applicable flags.

str
Required. The string on which to perform the search.

Remarks

The **test** method checks to see if a pattern exists within a string and returns **true** if so, and **false** otherwise.

The properties of the global **RegExp** object are not modified by the **test** method.

Example

The following example illustrates the use of the **test** method. To use this example, pass the function a regular expression pattern and a string. The function will test for the occurrence of the regular expression pattern in the string and return a string indicating the results of that search:

```
function TestDemo(re, s){
   var s1;                          //Declare variable.
   // Test string for existence of regular expression.
   if (re.test(s))                  //Test for existence.
      s1 = " contains ";            //s contains pattern.
   else
      s1 = " does not contain ";    //s does not contain pattern.
   return("'" + s + "'" + s1 + "'"+ re.source + "'"); //Return string.
}
```

Requirements

Version 3

See Also

RegExp Object | Regular Expression Object | Regular Expression Object Methods | Regular Expression Object Properties | Regular Expression Syntax

Applies To: Regular Expression Object

Build: Topic Version 5.6.9309.1546

JScript

toArray Method

Returns a standard JScript array converted from a VBArray.

safeArray.**toArray( )**

The required *safeArray* reference 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.

Example

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>
```

Requirements

Version 3

See Also

dimensions Method | getItem Method | lbound Method | ubound Method

Applies To: VBArray Object

Build: Topic Version 5.6.9309.1546

JScript

toDateString Method

Returns a date as a string value.

objDate.**toDateString( )**

The required *objDate* reference is a **Date** object.

Remarks

The **toDateString** method returns a string value containing the date, in the current time zone, in a convenient, easily read format.

Requirements

Version 5.5

See Also

toTimeString Method | toLocaleDateString Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

toExponential Method

Returns a string containing a number represented in exponential notation.

numObj.**toExponential(**[*fractionDigits*]**)**

Arguments

numObj
Required. A **Number** object.
fractionDigits
Optional. Number of digits after the decimal point. Must be in the range 0 – 20, inclusive.

Remarks

The **toExponential** method returns a string representation of a number in exponential notation. The string contains one digit before the significand's decimal point, and may contain *fractionDigits* digits after it.

If *fractionDigits* is not supplied, the **toExponential** method returns as many digits necessary to uniquely specify the number.

Requirements

Version 5.5

See Also

toFixed Method | toPrecision Method

Applies To: Number Object

Build: Topic Version 5.6.9309.1546

JScript

toFixed Method

Returns a string representing a number in fixed-point notation.

```
numObj.toFixed([fractionDigits])
```

Arguments

numObj
Required A **Number** object.
fractionDigits

Optional. Number of digits after the decimal point. Must be in the range 0 – 20, inclusive.

Remarks

The **toFixed** method returns a string representation of a number in fixed-point notation. The string contains one digit before the significand's decimal point, and must contain *fractionDigits* digits after it.

If *fractionDigits* is not supplied or **undefined**, the **toFixed** method assumes the value is zero.

Requirements

Version 5.5

See Also

toExponential Method | toPrecision Method

Applies To: Number Object

Build: Topic Version 5.6.9309.1546

JScript

toGMTString Method

Returns a date converted to a string using Greenwich Mean Time(GMT).

```
dateObj.toGMTString()
```

The required *dateObj* reference is any **Date** object.

Remarks

The **toGMTString** method is obsolete, and is provided for backwards compatibility only. It is recommended that you use the **toUTCString** method instead.

The **toGMTString** method returns a **String** object that contains the date formatted using GMT convention. The format of the return value is as follows: "05 Jan 1996 00:00:00 GMT."

Requirements

Version 1

See Also

Date Object Methods | toUTCString Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

toLocaleDateString Method

Returns a date as a string value appropriate to the host environment's current locale.

```
objDate.toLocaleDateString( )
```

The required *objDate* reference is a **Date** object.

Remarks

The **toLocaleDateString** method returns a string value that contains a date, in the current time zone, in an easily read format. The date is in the default format of the host environment's current locale. The return value of this method cannot be relied upon in scripting, as it will vary from computer to computer. The **toLocalDateString** method should only be used to format display – never as part of a computation.

Requirements

Version 5.5

See Also

toDateString Method | toLocaleTimeString Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

toLocaleLowerCase Method

Returns a string where all alphabetic characters have been converted to lowercase, taking into account the host environment's current locale.

```
stringVar.tolocaleLowerCase( )
```

The required *stringVar* reference is a **String** object, value, or literal.

Remarks

The **toLocaleLowerCase** method converts the characters in a string, taking into account the host environment's current locale. In most cases, the results are the same as you would obtain with the **toLowerCase** method. Results differ if the rules for a language conflict with the regular Unicode case mappings.

Requirements

Version 5.5

See Also

toLocaleUpperCase Method | toLowerCase Method

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

toLocaleString Method

Returns a date converted to a string using the current locale.

```
dateObj.toLocaleString()
```

The required *dateObj* is any **Date** object.

Remarks

The **toLocaleString** method returns a **String** object that contains the date written in the current locale's long default format.

- For dates between 1601 and 1999 A.D., the date is formatted according to the user's Control Panel Regional Settings.
- For dates outside this range, the default format of the **toString** method is used.

For example, in the United States, **toLocaleString** returns "01/05/96 00:00:00" for January 5. In Europe, it returns "05/01/96 00:00:00" for the same date, as European convention puts the day before the month.

> **Note** **toLocaleString** should only be used to display results to a user; it should never be used as the basis for computation within a script as the returned result is machine-specific.

Example

The following example illustrates the use of the **toLocaleString** method.

```
function toLocaleStrDemo(){
   var d, s;                     //Declare variables.
   d = new Date();               //Create Date object.
   s = "Current setting is ";
   s += d.toLocaleString();      //Convert to current locale.
   return(s);                    //Return converted date
}
```

Requirements

Version 1

See Also

Date Object Methods

Applies To: Array Object | Date Object | Number Object | Object Object

---

Build: Topic Version 5.6.9309.1546

JScript

toLocaleTimeString Method

Returns a time as a string value appropriate to the host environment's current locale.

objDate.**toLocaleTimeString( )**

The required *objDate* reference is a **Date** object.

Remarks

The **toLocaleTimeString** method returns a string value that contains a time, in the current time zone, in an easily read format. The time is in the default format of the host environment's current locale. The return value of this method cannot be relied upon in scripting, as it will vary from computer to computer. The **toLocalTimeString** method should only be used to format display – never as part of a computation.

Requirements

Version 5.5

See Also

ToTimeString Method | toLocaleDateString Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

toLocaleUpperCase Method

Returns a string where all alphabetic characters have been converted to uppercase, taking into account the host environment's current locale.

```
stringVar.tolocaleUpperCase( )
```

The required *stringVar* reference is a **String** object, value, or literal.

Remarks

The **toLocaleUpperCase** method converts the characters in a string, taking into account the host environment's current locale. In most cases, the results are the same as you would obtain with the **toUpperCase** method. Results differ if the rules for a language conflict with the regular Unicode case mappings.

Requirements

Version 5.5

See Also

toLocaleLowerCase Method | toUpperCase Method

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

toLowerCase Method

Returns a string where all alphabetic characters have been converted to lowercase.

```
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 object";
strVariable = strVariable.toLowerCase( );
```

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

```
this is a string object
```

Requirements

Version 1

See Also

String Object Methods | String Object Properties | toUpperCase Method

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

toPrecision Method

Returns a string containing a number represented either in exponential or fixed-point notation with a specified number of digits.

numObj.**toPrecision ([***precision***])**

Arguments

numObj
Required. A **Number** object.
precision
Optional. Number of significant digits. Must be in the range 1 – 21, inclusive.

Remarks

For numbers in exponential notation, *precision - 1* digits are returned after the decimal point. For numbers in fixed notation, *precision* significant digits are returned.

If precision is not supplied or is **undefined**, the **toString** method is called instead.

Requirements

Version 5.5

See Also

toFixed Method | toExponential Method

Applies To: Number Object

Build: Topic Version 5.6.9309.1546

JScript

toString Method

Returns a string representation of an object.

objectname.**toString(**[*radix*]**)**

Arguments

objectname
Required. An object for which a string representation is sought.

radix
Optional. Specifies a radix for converting numeric values to strings. This value is only used for numbers.

Remarks

The **toString** method is a member of all built-in JScript objects. How it behaves depends on the object type:

Object	Behavior
Array	Elements of an **Array** are converted to strings. The resulting strings are concatenated, separated by commas.
Boolean	If the Boolean value is **true**, returns "`true`". Otherwise, returns "`false`".
Date	Returns the textual representation of the date.
Error	Returns a string containing the associated error message.
Function	Returns a string of the following form, where *functionname* is the name of the function whose **toString** method was called: `function functionname( ) { [native code] }`
Number	Returns the textual representation of the number.
String	Returns the value of the **String** object.
Default	Returns "`[object objectname]`", where `objectname` is the name of the object type.

Example

The following example illustrates the use of the **toString** method with a radix argument. The return value of function shown below is a Radix

conversion table.

```
function CreateRadixTable (){
   var s, s1, s2, s3, x;                  //Declare variables.
   s = "Hex    Dec   Bin \n";             //Create table heading.
   for (x = 0; x < 16; x++)               //Establish size of table
   {                                      // in terms of number of
      switch(x)                           // values shown.
      {                                   //Set intercolumn spacing.
         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 = "    ";
      }                                   //Convert to hex, decimal & binary.
      s += " " + x.toString(16) + s1 + x.toString(10)
      s +=  s2 + s3 + x.toString(2)+ "\n";

   }
   return(s);                             //Return entire radix table.
}
```

Requirements

Version 2

See Also

function Statement

Applies To: Array Object | Boolean Object | Date Object | Error Object | Function Object | Number Object | Object Object | String Object

---

Build: Topic Version 5.6.9309.1546

JScript

toTimeString Method

Returns a time as a string value.

objDate.**toTimeString( )**

The required *objDate* reference is a **Date** object.

Remarks

The **toTimeString** method returns a string value containing the time, in the current time zone, in a convenient, easily read format.

Requirements

Version 5.5

See Also

toDateString Method | toLocaleTimeString Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

toUpperCase Method

Returns a string where all alphabetic characters have been converted to uppercase.

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

Remarks

The **toUpperCase** method has no effect on non-alphabetic characters.

Example

The following example demonstrates the effects of the **toUpperCase** method:

```
var strVariable = "This is a STRING object";
strVariable = strVariable.toUpperCase( );
```

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

```
THIS IS A STRING OBJECT
```

Requirements

Version 1

See Also

String Object Methods | String Object Properties | toLowerCase Method

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

toUTCString Method

Returns a date converted to a string using Universal Coordinated Time (UTC).

```
dateObj.toUTCString()
```

The required *dateObj* reference is any **Date** object.

Remarks

The **toUTCString** method returns a **String** object that contains the date formatted using UTC convention in a convenient, easily read form.

Example

The following example illustrates the use of the **toUTCString** method.

```
function toUTCStrDemo(){
   var d, s;                  //Declare variables.
   d = new Date();            //Create Date object.
   s = "Current setting is ";
   s += d.toUTCString();      //Convert to UTC string.
   return(s);                 //Return UTC string.
}
```

Requirements

Version 3

See Also

Date Object Methods | toGMTString Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

ubound Method

Returns the highest index value used in the specified dimension of the VBArray.

safeArray.**ubound(***dimension***)**

Arguments

safeArray
Required. A VBArray object.
dimension
Optional. The dimension of the VBArray for which the higher bound index is wanted. If omitted, **ubound** behaves as if a 1 was passed.

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.

Example

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 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(CreateVBArray()));
</SCRIPT>
</BODY>
```

Requirements

Version 3

See Also

dimensions Method | getItem Method | lbound Method | toArray Method

Applies To: VBArray Object

Build: Topic Version 5.6.9309.1546

JScript

unescape Method

Decodes **String** objects encoded with the **escape** method.

unescape(*charString*)

The required *charString* argument is a **String** object or literal to be decoded.

Remarks

The **unescape** method returns a string value 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 **%u***xxxx* format (Unicode characters) are replaced with the Unicode character with hexadecimal encoding *xxxx*.

> **Note** The **unescape** method should not be used to decode Uniform Resource Identifiers (URI). Use **decodeURI** and **decodeURIComponent** methods instead.

Requirements

Version 1

See Also

DecodeURI Method | decodeURIComponent Method | escape Method | String Object

Applies To: Global Object

Build: Topic Version 5.6.9309.1546

JScript

unshift Method

Returns an array with specified elements inserted at the beginning.

arrayObj.**unshift(**[*item1*[, *item2* [, . . . [, itemN]]]]**)**

Arguments

arrayObj
Required. An **Array** object.
item1, item2,. . ., itemN
Optional. Elements to insert at the start of the **Array**.

Remarks

The **unshift** method inserts elements into the start of an array, so they appear in the same order in which they appear in the argument list.

Requirements

Version 5.5

See Also

shift Method

Applies To: Array Object

Build: Topic Version 5.6.9309.1546

JScript

UTC Method

Returns the number of milliseconds between midnight, January 1, 1970 Universal Coordinated Time (UTC) (or GMT) and the supplied date.

```
Date.UTC(year, month, day[, hours[, minutes[, seconds[,ms]]]])
```

Arguments

year
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*.

month
Required. The month as an integer between 0 and 11 (January to December).

day
Required. The date as an integer between 1 and 31.

hours
Optional. Must be supplied if *minutes* is supplied. An integer from 0 to 23 (midnight to 11pm) that specifies the hour.

minutes
Optional. Must be supplied if *seconds* is supplied. An integer from 0 to 59 that specifies the minutes.

seconds
Optional. Must be supplied if *milliseconds* is supplied. An integer from 0 to 59 that specifies the seconds.

ms
Optional. An integer from 0 to 999 that specifies the milliseconds.

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.

> **Note** If *year* is between 0 and 99, use *1900 + year* for the year.

Example

The following example illustrates the use of the **UTC** method.

```
function DaysBetweenDateAndNow(yr, mo, dy){
   var d, r, t1, t2, t3;            //Declare variables.
   var MinMilli = 1000 * 60         //Initialize variables.
   var HrMilli = MinMilli * 60
   var DyMilli = HrMilli * 24
   t1 = Date.UTC(yr, mo - 1, dy)    //Get milliseconds since 1/1/1970.
   d = new Date();                  //Create Date object.
   t2 = d.getTime();                //Get current time.
   if (t2 >= t1)
      t3 = t2 - t1;
   else
      t3 = t1 - t2;
   r = Math.round(t3 / DyMilli);
   return(r);                       //Return difference.
}
```

Requirements

Version 1

See Also

Date Object Methods | setTime Method

Applies To: Date Object

Build: Topic Version 5.6.9309.1546

JScript

valueOf Method

Returns the primitive value of the specified object.

object.**valueOf( )**

The required *object* reference is any intrinsic JScript object.

Remarks

The **valueOf** method is defined differently for each intrinsic JScript object.

Object	Return Value
Array	The elements of the array are converted into strings, and the strings are concatenated together, separated by commas. This behaves the same as the **Array.toString** and **Array.join** methods.
Boolean	The Boolean value.
Date	The stored time value in milliseconds since midnight, January 1, 1970 UTC.
Function	The function itself.
Number	The numeric value.
Object	The object itself. This is the default.
String	The string value.

The **Math** and **Error** objects do not have a **valueOf** method.

Requirements

Version 2

See Also

toString Method

Applies To: Array Object | Boolean Object | Date Object | Function Object | Number Object | Object Object | String Object

---

Build: Topic Version 5.6.9309.1546

JScript

JScript Objects

The following table lists JScript Objects.

Description	Language Element
Enables and returns a reference to an Automation object.	ActiveXObject Object
Provides support for creation of arrays of any data type.	Array Object
Creates a new Boolean value.	Boolean Object
Enables basic storage and retrieval of dates and times.	Date Object
Object that stores data key, item pairs.	Dictionary Object
Enables enumeration of items in a collection.	Enumerator Object
An object that contains information about errors that occur while JScript code is running.	Error Object
Provides access to a computer's file system.	FileSystemObject Object
Creates a new function.	Function Object
An intrinsic object whose purpose is to collect global methods into one object.	Global Object
A intrinsic object that provides basic mathematics functionality and constants.	Math Object
An object representation of the number data type and placeholder for numeric constants.	Number Object

Provides functionality common to all JScript objects.	Object Object
Stores information on regular expression pattern searches.	RegExp Object
Contains a regular expression pattern.	Regular Expression Object
Allows manipulation and formatting of text strings and determination and location of substrings within strings.	String Object
Provides access to Visual Basic safe arrays.	VBArray Object

Build: Topic Version 5.6.9309.1546

JScript

ActiveXObject Object

Enables and returns a reference to an Automation object.

```
newObj = new ActiveXObject(servername.typename[, location])
```

Arguments

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

Remarks

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

document object, and a toolbar object.

To create an Automation object, assign the new **ActiveXObject** to an object variable:

```
var ExcelSheet;
ExcelApp = new ActiveXObject("Excel.Application");
ExcelSheet = new ActiveXObject("Excel.Sheet");
```

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.

```
// 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();
```

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 share name. For a network share named "\\myserver\public", the *servername* is "myserver". In addition, you can specify *servername* using DNS format or an IP address.

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

```
function GetAppVersion() {
   var XLApp = new ActiveXObject("Excel.Application", "MyServer");
   return(XLApp.Version);
}
```

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

Requirements

Version 1

See Also

GetObject Function

Build: Topic Version 5.6.9309.1546

JScript

Array Object

Provides support for creation of arrays of any data type.

```
arrayObj = new Array()
arrayObj = new Array([size])
arrayObj = new Array([element0[, element1[, ...[, elementN]]]])
```

Arguments

arrayObj
Required. The variable name to which the **Array** object is assigned.

size
Optional. The size of the array. As arrays are zero-based, created elements will have indexes from zero to *size* -1.

element0,...,elementN
Optional. The elements to place in the array. This creates an array with *n* + 1 elements, and a **length** of *n* + 1. Using this syntax, you must supply more than one element.

Remarks

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

```
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 must be an unsigned 32-bit integer (< approximately four billion). That value then becomes the size of the array. If the value is a number, but is less than zero or is not an integer, a run-time error occurs.

If a single value is passed to the **Array** constructor, and it is not a number, the **length** property is set to 1, and the value of the only element becomes the single, passed-in argument.

Notice that JScript arrays are sparse arrays, that is, although you can allocate an array with many elements, only the elements that actually contain data exist. This reduces the amount of memory used by the array.

Properties

constructor Property | length Property | prototype Property

Methods

concat Method | join Method | pop Method | push Method | reverse Method | shift Method | slice Method | sort Method | splice Method | toLocaleString Method | toString Method | unshift Method | valueOf Method

Requirements

Version 2

See Also

new Operator

Build: Topic Version 5.6.9309.1546

JScript

arguments Object

An object representing the arguments to the currently executing function, and the functions that called it.

[*function*.]**arguments[***n***]**

Arguments

function
Optional. The name of the currently executing **Function** object.

n
Required. The zero-based index to argument values passed to the **Function** object.

Remarks

You cannot explicitly create an **arguments** object. The **arguments** object only becomes available when a function begins execution. The **arguments** object of the function is not an array, but the individual arguments are accessed the same way array elements are accessed. The index *n* is actually a reference to one of the **0…*n*** properties of the **arguments** object.

Example

The following example illustrates the use of the **arguments** object.

```
function ArgTest(a, b){
   var i, s = "The ArgTest function expected ";
   var numargs = arguments.length;     //Get number of arguments passed.
   var expargs = ArgTest.length;       //Get number of arguments expected.
   if (expargs < 2)
```

```
      s += expargs + " argument. ";
   else
      s += expargs + " arguments. ";
   if (numargs < 2)
      s += numargs + " was passed.";
   else
      s += numargs + " were passed.";
   s += "\n\n"
   for (i =0 ; i < numargs; i++){      //Get argument contents.
   s += "  Arg " + i + " = " + arguments[i] + "\n";
   }
   return(s);                          //Return list of arguments.
}
```

Requirements

Version 1

See Also

0…n Properties | callee Property | length Property

Build: Topic Version 5.6.9309.1546

JScript

Boolean Object

Creates a new Boolean value.

Syntax

```
boolObj = new Boolean([boolValue])
```

Arguments

boolObj
Required. The variable name to which the **Boolean** object is assigned.
boolValue
Optional. The initial Boolean value for the new object. If *Boolvalue* 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.

Properties

constructor Property | prototype Property

Methods

toString Method | valueOf Method

Requirements

Version 2

See Also

new Operator | var Statement

Build: Topic Version 5.6.9309.1546

JScript

Date Object

Enables basic storage and retrieval of dates and times.

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

Arguments

dateObj
Required. The variable name to which the **Date** object is assigned.

dateVal
Required. If a numeric value, *dateVal* represents the number of milliseconds in Universal Coordinated Time between the specified date and midnight January 1, 1970. If a string, *dateVal* is parsed according to the rules in the **parse** method. The *dateVal* argument can also be a VT_DATE value as returned from some ActiveX® objects.

year
Required. The full year, for example, 1976 (and not 76).

month
Required. The month as an integer between 0 and 11 (January to December).

date
Required. The date as an integer between 1 and 31.

hours
Optional. Must be supplied if *minutes* is supplied. An integer from 0 to 23 (midnight to 11pm) that specifies the hour.

minutes
Optional. Must be supplied if *seconds* is supplied. An integer from 0 to 59 that specifies the minutes.

seconds
Optional. Must be supplied if *milliseconds* is supplied. An integer from 0 to 59 that specifies the seconds.

ms
Optional. An integer from 0 to 999 that specifies the milliseconds.

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**, 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**.

Error

The following example illustrates the use of the **Date** object.

```
function DateDemo(){
   var d, s = "Today's date is: ";          //Declare variables.
   d = new Date();                          //Create Date object.
   s += (d.getMonth() + 1) + "/";           //Get month
   s += d.getDate() + "/";                  //Get day
   s += d.getYear();                        //Get year.
   return(s);                               //Return date.
}
```

Properties

constructor Property | prototype Property

Methods

getDate Method | getDay Method | getFullYear Method | getHours Method | getMilliseconds Method | getMinutes Method | getMonth Method | getSeconds Method | getTime Method | getTimezoneOffset Method | getUTCDate Method | getUTCDay Method | getUTCFullYear Method | getUTCHours Method | getUTCMilliseconds Method | getUTCMinutes Method | getUTCMonth Method | getUTCSeconds Method | getVarDate Method | 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 | parse Method | UTC Method

Requirements

Version 1

See Also

new Operator | var Statement

Build: Topic Version 5.6.9309.1546

JScript

Enumerator Object

Enables enumeration of items in a collection.

```
enumObj = new Enumerator([collection])
```

Arguments

enumObj
Required. The variable name to which the **Enumerator** object is assigned.
collection
Optional. 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.

Example

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

```
function ShowDriveList(){
   var fso, s, n, e, x;                    //Declare variables.
   fso = new ActiveXObject("Scripting.FileSystemObject");
   e = new Enumerator(fso.Drives);         //Create Enumerator on Drives.
   s = "";
   for (;!e.atEnd();e.moveNext())          //Enumerate drives collection.
      {
         x = e.item();
         s = s + x.DriveLetter;
         s += " - ";
         if (x.DriveType == 3)             //See if network drive.
            n = x.ShareName;               //Get share name
         else if (x.IsReady)               //See if drive is ready.
            n = x.VolumeName;              //Get volume name.
         else
            n = "[Drive not ready]";
         s +=  n + "<br>";
      }
   return(s);                              //Return active drive list.
}
```

Properties

The **Enumerator** object has no properties.

Methods

atEnd Method | item Method | moveFirst Method | moveNext Method

Requirements

Version 3

Build: Topic Version 5.6.9309.1546

JScript

Error Object

Contains information about errors.

```
errorObj = new Error()
errorObj = new Error([number])
errorObj = new Error([number[, description]])
```

Arguments

errorObj
Required. The variable name to which the **Error** object is assigned.
number
Optional. Numeric value assigned to an error. Zero if omitted.
description
Optional. Brief string that describes an error. Empty string if omitted.

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 is 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.

Example

The following example illustrates the use of the implicitly created **Error** object.

```
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".
}
```

Methods

The **Error** object has no methods.

Properties

description Property | number Property

Requirements

Version 5

See Also

new Operator | throw Statement | try...catch Statement | var Statement

---

Build: Topic Version 5.6.9309.1546

JScript

Function Object

Creates a new function.

Syntax 1

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

Syntax 2

```
functionName = new Function( [argname1, [... argnameN,]] body );
```

Arguments

functionName
Required. The name of the newly created function

argname1...argnameN
Optional. A list of arguments the function accepts.

body
Optional. A string that contains the block of JScript code to be executed when the function is called.

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

```
function add(x, y)
{
   return(x + y);            //Perform addition and return results.
}
```

Example 2

```
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:

```
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.

Properties

arguments Property | caller Property | constructor Property | prototype Property

Methods

toString Method | valueOf Method

Requirements

Version 2

See Also

function Statement | new Operator | var Statement

Build: Topic Version 5.6.9309.1546

JScript

Global Object

An intrinsic object whose purpose is to collect global methods into one object.

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.

Properties

Infinity Property | NaN Property

Methods

escape Method | eval Method | isFinite Method | isNaN Method | parseFloat Method | parseInt Method | unescape Method

Requirements

Version 5

See Also

Object Object

Build: Topic Version 5.6.9309.1546

JScript

Math Object

An intrinsic object that provides basic mathematics functionality and constants.

```
Math.[{property | method}]
```

Arguments

property
Required. Name of one of the properties of the **Math**. object.
method
Required. Name of one of the methods of the **Math**. object.

Remarks

The **Math** object cannot be created using the **new** operator, and gives an error if you attempt to do so. The scripting engine creates it when the engine is loaded. All of its methods and properties are available to your script at all times.

Properties

E Property | LN2 Property | LN10 Property | LOG2E Property | LOG10E Property | PI Property | SQRT1_2 Property | SQRT2 Property

Methods

abs Method | acos Method | asin Method | atan Method | atan2 Method | ceil Method | cos Method | exp Method | floor Method | log Method | max Method | min Method | pow Method | random Method | round Method | sin Method | sqrt Method | tan Method

Requirements

Version 1

See Also

Number Object

Build: Topic Version 5.6.9309.1546

JScript

Number Object

An object representation of the number data type and placeholder for numeric constants.

numObj = **new Number(***value***)**

Arguments

numobj
Required. The variable name to which the **Number** object is assigned.

value
Required. The numeric value of the **Number** object being created.

Remarks

JScript creates **Number** objects as required from numerical values. It is rarely necessary to create **Number** objects explicitly.

The primary purposes for the **Number** object are to collect its properties into one object, and to allow numbers to be converted into strings via the **toString** method.

Properties

MAX_VALUE Property | MIN_VALUE Property | NaN Property | NEGATIVE_INFINITY Property | POSITIVE_INFINITY Property | constructor Property | prototype Property

Methods

toLocaleString Method | toString Method | valueOf Method

Requirements

Version 1

See Also

Math Object | new Operator

Build: Topic Version 5.6.9309.1546

JScript

Object Object

Provides functionality common to all JScript objects.

```
obj = new Object([value])
```

Arguments

obj
Required. The variable name to which the **Object** object is assigned.
value

Optional. Any one of the JScript primitive data types (Number, Boolean, or String. If value is an object, the object is returned unmodified. If *value* is **null**, **undefined**, or not supplied, an object with no content 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.

Properties

prototype Property | constructor Property

Methods

toLocaleString Method | toString Method | valueOf Method

Requirements

Version 3

See Also

Function Object | Global Object

Build: Topic Version 5.6.9309.1546

JScript

RegExp Object

An intrinsic global object that stores information about the results of regular expression pattern matches.

RegExp.*property*

The required *property* argument can be any one of the **RegExp** object properties.

Remarks

The **RegExp** object cannot be created directly, but is always available for use. Until a successful regular expression search has been completed, the initial values of the various properties of the RegExp object are as follows:

Property	Shorthand	Initial Value
index		-1
lastIndex		-1
lastMatch	$&	Empty string.
lastParen	$+	Empty string.
leftContext		Empty string.
rightContext		Empty string.
$1 - $9	$1 - $9	Empty string.

Its properties have undefined as their value until a successful regular expression search has been completed.

The global **RegExp** object should not be confused with the **Regular Expression** object. Even though they sound like the same thing, they are separate and distinct. The properties of the global **RegExp** object contain continually updated information about each match as it occurs, while the properties of the **Regular Expression** object contain only information about the matches that occur with that instance of the **Regular Expression**.

Example

The following example illustrates the use of the global **RegExp** object.

```
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 + "\n";
   s += "$2 contains: " + RegExp.$2 + "\n";
   s += "$3 contains: " + RegExp.$3;
   return(s);
}
```

Properties

$1...$9 Properties | index Property | input Property | lastIndex Property | lastMatch Property | lastParen Property | leftContext Property | rightContext Property

Methods

The **RegExp** object has no methods.

Requirements

Version 3

See Also

Regular Expression Object | Regular Expression Syntax | String Object

---

Build: Topic Version 5.6.9309.1546

JScript

Regular Expression Object

An object that contains a regular expression pattern along with flags that identify how to apply the pattern.

Syntax 1

```
re = /pattern/[flags]
```

Syntax 2

```
re = new RegExp("pattern",["flags"])
```

Arguments

re
Required. The variable name to which the regular expression pattern is assigned.

pattern
Required. The regular expression pattern to use. If you use Syntax 1, delimit the pattern by "/" characters. If you use Syntax 2, enclose the pattern in quotation marks.

flags
Optional. Enclose flag in quotation marks if you use Syntax 2. Available flags, which may be combined, are:

- g (global search for all occurrences of *pattern*)
- i (ignore case)
- m (multiline search)

Remarks

The **Regular Expression** object should not be confused with the global **RegExp** object. Even though they sound the same, they are separate and distinct. The properties of the **Regular Expression** object contain only information about one particular **Regular Expression** instance, while the properties of the global **RegExp** object contain continually updated information about each match as it occurs.

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 global **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.

Example

The following example illustrates the use of the **Regular Expression** object by creating an object (re) containing a regular expression pattern with its associated flags. In this case, the resulting **Regular Expression** object is then used by the **match** method:

```
function MatchDemo(){
   var r, re;                    //Declare variables.
   var s = "The rain in Spain falls mainly in the plain";
   re = new RegExp("Spain","i");  //Create regular expression object.
   r = s.match(re);               //Find a match within string s.
   return(r);                    //Return results of match.
}
```

Properties

lastIndex Property | source Property

Methods

compile Method | exec Method | test Method

Requirements

Version 3

See Also

RegExp Object | Regular Expression Syntax | String Object

Build: Topic Version 5.6.9309.1546

JScript

String Object

Allows manipulation and formatting of text strings and determination and location of substrings within strings.

Syntax

newString = **new String(**["s*tringLiteral*"]**)**

Arguments

newString
Required. The variable name to which the **String** object is assigned.

stringLiteral
Optional. Any group of Unicode characters.

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. If a property is added to a string literal, it is available to all standard string objects:

```
var alpha, beta;
alpha = "This is a string";
beta = "This is also a string";

alpha.test = 10;
```

In the previous example, *test* is now defined for *beta* and all future string literals. In the following example, however, added properties are treated differently:

```
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.

Properties

constructor Property | length Property | prototype Property

Methods

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 | sub Method | substr Method | substring Method | sup Method | toLowerCase Method | toUpperCase Method | toString Method | valueOf Method

Requirements

Version 1

See Also

new Operator

Build: Topic Version 5.6.9309.1546

JScript

VBArray Object

Provides access to Visual Basic safe arrays.

```
varName = new VBArray(safeArray)
```

Arguments

varName
Required. The variable name to which the VBArray is assigned.
safeArray
Required. 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.

Example

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("vbCRLF")
   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>
```

Properties

The VBArray object has no properties.

Methods

dimensions Method | getItem Method | lbound Method | toArray Method | ubound Method

Requirements

Version 3

See Also

Array Object

Build: Topic Version 5.6.9309.1546

JScript

JScript Operators

The following table lists JScript operators

Description	Language Element
Sums two numbers or concatenates two strings.	Addition Operator (+)
Assigns a value to a variable.	Assignment Operator (=)
Performs a bitwise AND on two expressions.	Bitwise AND Operator (&)
Shifts the bits of an expression to the left.	Bitwise Left Shift Operator (<<)
Performs a bitwise NOT (negation) on an expression.	Bitwise NOT Operator (~)
Performs a bitwise OR on two expressions.	Bitwise OR Operator (|)
Shifts the bits of an expression to the right, maintaining sign.	Bitwise Right Shift Operator (>>)
Performs a bitwise exclusive OR on two expressions.	Bitwise XOR Operator (^)
Causes two expressions to be executed sequentially.	Comma Operator (,)
Returns a Boolean value indicating the result of the comparison.	Comparison Operators
List of compound assignment operators.	Compound Assignment Operators
Executes one of two expressions depending on a condition.	Conditional (ternary) Operator (?:)
Decrements a variable by one.	Decrement Operator (—)
Deletes a property from an object, or removes an element from an array.	delete Operator
Divides two numbers and returns a numeric result.	Division Operator (/)
Compares two expressions to determine if they are equal.	Equality Operator (==)
Compares two expressions to determine if one is greater than the	Greater than Operator (>)

other.	
Compares two expressions to determine if one is greater than or equal to the other.	Greater than or equal to Operator (>=)
Compares two expressions to determine if they are equal in value and of the same data type.	Identity Operator (===)
Increments a variable by one.	Increment Operator (++)
Compares two expressions to determine if they are unequal.	Inequality Operator (!=)
Returns a Boolean value that indicates whether or not an object is an instance of a particular class.	instanceof Operator
Compares two expressions to determine if one is less than the other.	Less than Operator (<)
Compares two expressions to determine if one is less than or equal to the other.	Less than or equal to Operator (<=)
Performs a logical conjunction on two expressions.	Logical AND Operator (&&)
Performs logical negation on an expression.	Logical NOT Operator (!)
Performs a logical disjunction on two expressions.	Logical OR Operator (||)
Divides two numbers and returns the remainder.	Modulus Operator (%)
Multiplies two numbers.	Multiplication Operator (*)
Creates a new object.	new Operator
Compares two expressions to determine that they are not equal in value or of the same data type.	Nonidentity Operator (!==)
List containing information about the execution precedence of JScript operators.	Operator Precedence
Performs subtraction of two expressions.	Subtraction Operator (-)
Returns a string that identifies the data type of an expression.	typeof Operator
Indicates the negative value of a numeric expression.	Unary Negation Operator (-)
Performs an unsigned right shift of the bits in an expression.	Unsigned Right Shift Operator (>>>)
Prevents an expression from returning a value.	void Operator

Build: Topic Version 5.6.9309.1546

JScript

Addition Assignment Operator (+=)

Adds the value of an expression to the value of a variable and assigns the result to the variable.

```
result += expression
```

Arguments

result
Any variable.
expression
Any expression.

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.

If	Then
Both expressions are numeric or Boolean	Add
Both expressions are strings	Concatenate
One expression is numeric and the other is a string	Concatenate

Requirements

Version 1

See Also

+ Operator | Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Addition Operator (+)

Adds the value of one numeric expression to another or concatenates two strings.

```
result = expression1 + expression2
```

Arguments

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

Remarks

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

If	Then
Both expressions are numeric or Boolean	Add
Both expressions are strings	Concatenate
One expression is numeric and the other is a string	Concatenate

Requirements

Version 1

See Also

+= Operator | Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Assignment Operator (=)

Assigns a value to a variable.

```
result = expression
```

Arguments

result
Any variable.
expression
Any numeric expression.

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`, and `l` equal zero after the example statement is executed.

Requirements

Version 1

See Also

Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Bitwise AND Assignment Operator (&=)

Performs a bitwise AND on the value of a variable and the value of an expression and assigns the result to the variable.

```
result &= expression
```

Arguments

result
Any variable.
expression
Any expression.

Remarks

Using this operator is exactly the same as specifying:

```
result = result & expression
```

The **&=** 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:

```
0101    (result)
1100    (expression)
----
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.

Requirements

Version 1

See Also

& Operator | Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Bitwise AND Operator (&)

Performs a bitwise AND on two expressions.

result = *expression1* & *expression2*

Arguments

result
Any variable.
expression1
Any expression.

expression2
Any expression.

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.

Requirements

Version 1

See Also

&= Operator | Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Bitwise Left Shift Operator (<<)

Left shifts the bits of an expression.

```
result = expression1 << expression2
```

Arguments

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

Remarks

The << operator shifts the bits of *expression1* left by the number of bits specified in *expression2*. For example:

```
var temp
temp = 14 << 2
```

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

Requirements

Version 1

See Also

<<= Operator | >> Operator | >>> Operator | Operator Precedence | Operator Summary

---

Build: Topic Version 5.6.9309.1546

JScript

Bitwise NOT Operator (~)

Performs a bitwise NOT (negation) on an expression.

```
result = ~ expression
```

Arguments

result
Any variable.
expression
Any expression.

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.

Requirements

Version 1

See Also

! Operator | Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Bitwise OR Assignment Operator (|=)

Performs a bitwise OR on the value of a variable and the value of an expression and assigns the result to the variable.

```
result |= expression
```

Arguments

result
Any variable.
expression
Any expression.

Remarks

Using this operator is exactly the same as specifying:

```
result = result | expression
```

The |= operator looks at the binary representation of the values of *result* and *expression* and does a bitwise OR operation on them. The result of this operation behaves like this:

```
0101    (result)
```

```
1100    (expression)
----
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.

Requirements

Version 1

See Also

| Operator | Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Bitwise OR Operator (|)

Performs a bitwise OR on two expressions.

result = *expression1* | *expression2*

Arguments

result
Any variable.
expression1
Any expression.
expression2

Any expression.

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:

```
0101   (expression1)
1100   (expression2)
----
1101   (result)
```

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.

Requirements

Version 1

See Also

|= Operator | Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Bitwise Right Shift Operator (>>)

Right shifts the bits of an expression, maintaining sign.

```
result = expression1 >> expression2
```

Arguments

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

Remarks

The >> 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 (11111100 in binary).

```
var temp
temp = -14 >> 2
```

Requirements

Version 1

See Also

<< Operator | >>= Operator | >>> Operator | Operator Precedence | Operator Summary

---

Build: Topic Version 5.6.9309.1546

JScript

Bitwise XOR Assignment Operator (^=)

Performs a bitwise exclusive OR on a variable and an expression and assigns the result to the variable.

```
result ^= expression
```

Arguments

result
Any variable.
expression
Any expression.

Remarks

Using the ^= operator is exactly the same as specifying:

```
result = result ^ 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:

```
0101    (result)
1100    (expression)
----
1001    (result)
```

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.

Requirements

Version 1

See Also

^ Operator | Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Bitwise XOR Operator (^)

Performs a bitwise exclusive OR on two expressions.

```
result = expression1 ^ expression2
```

Arguments

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

Remarks

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:

```
0101   (expression1)
1100   (expression2)
----
1001   (result)
```

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.

Requirements

Version 1

See Also

^= Operator | Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Comma Operator (,)

Causes two expressions to be executed sequentially.

expression1, expression2

Arguments

expression1
Any expression.
expression2
Any expression.

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:

```
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.

Requirements

Version 1

See Also

for Statement | Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Comparison Operators

Returns a Boolean value indicating the result of the comparison.

expression1 **comparisonoperator** *expression2*

Arguments

expression1
Any expression.
comparisonoperator
Any comparison operator.
expression2
Any expression.

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.

Requirements

Version 1

See Also

Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Compound Assignment Operators

The following table lists JScript assignment operators.

Operator	Symbol
Addition	+=
Bitwise AND	&=
Bitwise Or	|=
Bitwise XOR	^=
Division	/=
Left Shift	<<=
Modulus	%=
Multiplication	*=
Right Shift	>>=
Subtraction	-=
Unsigned Right Shift	>>>=

Requirements

Version Information

Build: Topic Version 5.6.9309.1546

JScript

Conditional (Ternary) Operator (?:)

Executes one of two statements depending on a condition.

```
test ? statement1 : statement2
```

Arguments

test
Any Boolean expression.
statement1
A statement executed if *test* is **true**. May be a compound statement.
statement2
A statement executed if *test* is **false**. May be a compound statement.

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:

```
var now = new Date();
var greeting = "Good" + ((now.getHours() > 17) ? " evening." : " day.");
```

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:

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

Requirements

Version 1

See Also

if...else Statement | Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

delete Operator

Deletes a property from an object, or removes an element from an array.

```
delete expression
```

The *expression* argument is a valid JScript expression that usually results 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.

Requirements

Version 3

See Also

Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Division Assignment Operator (/=)

Divides the value of a variable by the value of an expression and assigns the result to the variable.

```
result /= expression
```

Arguments

result
Any numeric variable.
expression
Any numeric expression.

Remarks

Using the **/=** operator is exactly the same as specifying:

```
result = result / expression
```

Requirements

Version 1

See Also

/ Operator | Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Division Operator (/)

Divides the value of two expressions.

```
result = number1 / number2
```

Arguments

result
Any numeric variable.
number1
Any numeric expression.
number2
Any numeric expression.

Requirements

Version 1

See Also

/= Operator | Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

in Operator

Tests for the existence of a property in an object.

```
result = property in object
```

Arguments

result
Required. Any variable.
property
Required. An expression that evaluates to a string expression.
object
Required. Any object.

Remarks

The **in** operator checks if an object has a property named property. It also checks the object's prototype to see if the property is part of the prototype chain.

Requirements

Version 1

See Also

Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Increment (++) and Decrement (—) Operators

Increments or decrements the value of a variable by one.

Syntax 1

```
result = ++variable
result = --variable
result = variable++
result = variable--
```

Syntax 2

```
++variable
--variable
variable++
variable--
```

Arguments

result
Any variable.
variable
Any variable.

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:

```
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:

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

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

Requirements

Version 1

See Also

Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

instanceof Operator

Returns a Boolean value that indicates whether or not an object is an instance of a particular class.

```
result = object instanceof class
```

Arguments

result
Required. Any variable.
object
Required. Any object expression.
class
Required. Any defined object class.

Remarks

The **instanceof** operator returns **true** if *object* is an instance of *class*. It returns **false** if *object* is not an instance of the specified class, or if *object* is **null**.

Example

The following example illustrates the use of the **instanceof** operator.

```
function objTest(obj){
   var i, t, s = "";   // Create variables.
   t = new Array();   // Create an array.
   t["Date"] = Date;   // Populate the array.
   t["Object"] = Object;
   t["Array"] = Array;
      for (i in t)
      {
         if (obj instanceof t[i])   // Check class of obj.
         {
            s += "obj is an instance of " + i + "\n";
         }
         else
         {
            s += "obj is not an instance of " + i + "\n";
         }
   }
   return(s);   // Return string.
}
```

```
var obj = new Date();
response.write(objTest(obj));
```

Requirements

Version 5

See Also

Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Left Shift Assignment Operator (<<=)

Left shifts the value of a variable by the number of bits specified in the value of an expression and assigns the result to the variable.

result **<<=** *expression*

Arguments

result
Any variable.
expression
Any expression.

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.

Requirements

Version 1

See Also

<< Operator | >> Operator | >>> Operator | Operator Precedence | Operator Summary

---

Build: Topic Version 5.6.9309.1546

JScript

Logical AND Operator (&&)

Performs a logical conjunction on two expressions.

```
result = expression1 && expression2
```

Arguments

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

Remarks

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

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.

Requirements

Version 1

See Also

Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Logical NOT Operator (!)

Performs logical negation on an expression.

result = !*expression*

Arguments

result
Any variable.
expression
Any expression.

Remarks

The following table illustrates how *result* is determined.

If `expression` is	Then `result` is
True	False
False	True

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.

Requirements

Version 1

See Also

~ Operator | Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Logical OR Operator (||)

Performs a logical disjunction on two expressions.

```
result = expression1 || expression2
```

Arguments

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

Remarks

If either or both expressions evaluate to **True**, *result* is **True**. The following table illustrates how *result* is determined:

If `expression1` is	And `expression2` is	The `result` is
True	True	True
True	False	True

False	True	True
False	False	False

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.

Requirements

Version 1

See Also

Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Modulus Assignment Operator (%=)

Divides the value of a variable by the value of an expression, and assigns the remainder to the variable.

```
result %= expression
```

Arguments

result
Any variable.
expression
Any numeric expression.

Remarks

Using the **%=** operator is exactly the same as specifying:

```
result = result % expression
```

Requirements

Version 1

See Also

% Operator | Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Modulus Operator (%)

Divides the value of one expression by the value of another, and returns the remainder.

```
result = number1 % number2
```

Arguments

result
Any variable.
number1
Any numeric expression.
number2
Any numeric expression.

Remarks

The modulus, or remainder, operator divides *number1* by *number2* (rounding floating-point numbers to integers) and returns only the remainder as *result*. For example, in the following expression, A (which is *result*) equals 5.

```
A = 19 % 6.7
```

Requirements

Version 1

See Also

%= Operator | Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Multiplication Assignment Operator (*=)

Multiplies the value of a variable by the value of an expression and assigns the result to the variable.

```
result *= expression
```

Arguments

result
Any variable.
expression
Any expression.

Remarks

Using the ***=** operator is exactly the same as specifying:

```
result = result * expression
```

Requirements

Version 1

See Also

* Operator | Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Multiplication Operator (*)

Multiplies the value of two expressions.

```
result = number1*number2
```

Arguments

result
Any variable.
number1
Any expression.
number2
Any expression.

Requirements

Version 1

See Also

*= Operator | Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

new Operator

Creates a new object.

```
new constructor[(arguments)]
```

Arguments

constructor
Required. Object's constructor. The parentheses can be omitted if the construc

tor takes no arguments.

arguments
Optional. Any arguments to be passed to the new object's constructor.

Remarks

The **new** operator performs the following tasks:

- It creates an object with no members.
- 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.

```
my_object = new Object;
my_array = new Array();
my_date = new Date("Jan 5 1996");
```

Requirements

Version 1

See Also

function Statement

Build: Topic Version 5.6.9309.1546

JScript

Right Shift Assignment Operator (>>=)

Right shifts the value of a variable by the number of bits specified in the value of an expression, maintaining the sign, and assigns the result to the variable.

```
result >>= expression
```

Arguments

result
Any variable.
expression
Any expression.

Remarks

Using the >>= operator is exactly the same as specifying:

```
result = result >> expression
```

The >>= 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
```

Requirements

Version 1

See Also

<< Operator | >> Operator | >>> Operator | Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Subtraction Assignment Operator (-=)

Subtracts the value of an expression from the value of a variable and assigns the result to the variable.

```
result -= expression
```

Arguments

result
Any numeric variable.
expression
Any numeric expression.

Remarks

Using the **-=** operator is exactly the same as doing the following:

```
result = result – expression
```

Requirements

Version 1

See Also

- Operator | Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Subtraction Operator (-)

Subtracts the value of one expression from another or provides unary negation of a single expression.

Syntax 1

```
result = number1 - number2
```

Syntax 2

```
-number
```

Arguments

result
Any numeric variable.
number
Any numeric expression.
number1
Any numeric expression.
number2
Any numeric expression.

Remarks

In Syntax 1, the - operator is the arithmetic subtraction operator used to find the difference between two numbers. In Syntax 2, the - operator is used as the unary negation operator to indicate the negative value of an expression.

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.

Requirements

Version 1

See Also

-= Operator | Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

typeof Operator

Returns a string that identifies the data type of an expression.

```
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.

Requirements

Version 1

See Also

Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

Unsigned Right Shift Operator (>>>)

Right shifts the bits of an expression, without maintaining sign.

```
result = expression1 >>> expression2
```

Arguments

result
Any variable.
expression1

Any expression.

expression2

Any expression.

Remarks

The >>> operator shifts the bits of *expression1* right by the number of bits specified in *expression2*. Zeroes are filled in from the left. Digits shifted off the right are discarded. For example:

```
var temp
temp = -14 >>> 2
```

The variable *temp* has a value of 1073741820 as -14 (11111111 11111111 11111111 11110010 in binary) shifted right two bits equals 1073741820 (00111111 11111111 11111111 11111100 in binary).

Requirements

Version 1

See Also

>>>= Operator | << Operator | >> Operator | Operator Precedence | Operator Summary

---

Build: Topic Version 5.6.9309.1546

JScript

Unsigned Right Shift Assignment Operator (>>>=)

Right shifts the value of a variable by the number of bits specified in the value of an expression, without maintaining sign, and assigns the result to the variable.

result **>>>=** *expression*

Arguments

result
Any variable.
expression
Any expression.

Remarks

Using the >>>= operator is exactly the same as doing the following:

```
result = result >>> expression
```

The **>>>=** 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 11111111 11111111 11110010 in binary) shifted right two bits equals 1073741820 (00111111 11111111 11111111 11111100 in binary).

Requirements

Version 1

See Also

>>> Operator | << Operator | >> Operator | Operator Precedence | Operator Summary

---

Build: Topic Version 5.6.9309.1546

JScript

void Operator

Prevents an expression from returning a value.

```
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.

Requirements

Version 2

See Also

Operator Precedence | Operator Summary

Build: Topic Version 5.6.9309.1546

JScript

JScript Properties

The following table lists JScript properties.

Description	Language Element
Returns the nine most-recently memorized portions found during pattern matching.	$1...$9 Properties
Returns an array containing each argument passed to the currently executing function.	arguments Property
Returns a reference to the function that invoked the current function.	caller Property
Specifies the function that creates an object.	constructor Property
Returns or sets the descriptive string associated with a specific error.	description Property
Returns Euler's constant, the base of natural logarithms.	E Property
Returns the character position where the first successful match begins in a searched string.	index Property
Returns an initial value of **Number.POSITIVE_INFINITY**.	Infinity Property
Returns the string against which a search was performed.	input Property
Returns the character position where the last successful match begins in a searched string.	lastIndex Property
Returns an integer value one higher than the highest element defined in an array.	length Property (Array)
Returns the number of arguments defined for a function.	length Property (Function)
Returns the length of a **String** object.	length Property (String)
Returns the natural logarithm of 2.	LN2 Property
Returns the natural logarithm of 10.	LN10 Property
Returns the base-2 logarithm of *e*, Euler's constant.	LOG2E Property
Returns the base-10 logarithm of *e*, Euler's constant.	LOG10E Property
Returns the largest number that can be represented in JScript.	MAX_VALUE Property
Returns the number closest to zero that can be represented in JScript.	MIN_VALUE Property
Returns the special value **NaN** indicating that an expression is not a number.	NaN Property (Global)
Returns the special value (**NaN**) indicating that an expression is not a number.	NaN Property (Number)
Returns a value more negative than the largest negative number (-	NEGATIVE_INFINITY Property

Number.MAX_VALUE) that can be represented in JScript.	
Returns or sets the numeric value associated with a specific error.	number Property
Returns the ratio of the circumference of a circle to its diameter, approximately 3.141592653589793.	PI Property
Returns a value larger than the largest number (Number.MAX_VALUE) that can be represented in JScript.	POSITIVE_INFINITY Property
Returns a reference to the prototype for a class of objects.	prototype Property
Returns a copy of the text of the regular expression pattern.	source Property
Returns the square root of 0.5, or one divided by the square root of 2.	SQRT1_2 Property
Returns the square root of 2.	SQRT2 Property

Build: Topic Version 5.6.9309.1546

JScript

0...*n* Properties

Returns the actual value of individual arguments from an **arguments** object returned by the **arguments** property of an executing function.

```
[function.]arguments[[0|1|2|...|n]]
```

Arguments

function
Optional. The name of the currently executing **Function** object.

0, 1, 2, …, *n*
Required. Non-negative integer in the range of 0 to *n* where 0 represents the first argument and *n* represents the final argument. The value of the final argument *n* is **arguments.length-1**.

Remarks

The values returned by the 0 . . . n properties are the actual values passed to the executing function. While not actually an array of arguments, the individual arguments that comprise the **arguments** object are accessed the same way that array elements are accessed.

Example

The following example illustrates the use of the **0 . . . *n*** properties of the **arguments** object. To fully understand the example, pass one or more arguments to the function:

```
function ArgTest(){
   var s = "";
   s += "The individual arguments are: "
   for (n=0; n< arguments.length; n++){
      s += ArgTest.arguments[n];
      s += " ";
   }
   return(s);
}
print(ArgTest(1, 2, "hello", new Date()));
```

Requirements

Version 5.5

See Also

Applies To: arguments Object | Function object

Build: Topic Version 5.6.9309.1546

JScript

$1...$9 Properties

Returns the nine most-recently memorized portions found during pattern matching. Read-only.

RegExp.$*n*

Arguments

RegExp
Always the global **RegExp** object.

n

Any integer from 1 through 9.

Remarks

The value of the **$1...$9** properties is modified whenever a successful parenthesized match is made. Any number of parenthesized substrings may be specified in a regular expression pattern, but only the nine most recent can be stored.

The following example illustrates the use of the **$1...$9** properties:

```
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 + "\n";
   s += "$2 contains: " + RegExp.$2 + "\n";
   s += "$3 contains: " + RegExp.$3;
   return(s);
}
```

Requirements

Version 1

See Also

RegExp Object Properties | Regular Expression Syntax

Applies To: RegExp Object

Build: Topic Version 5.6.9309.1546

JScript

arguments Property

Returns the **arguments** object for the currently executing **Function** object.

function.**arguments**

The *function* argument is the name of the currently executing function, and can be omitted.

Remarks

The **arguments** property allows a function to handle a variable number of arguments. The **length** property of the **arguments** object contains the number of arguments passed to the function. The individual arguments contained in the **arguments** object can be accessed in the same way array elements are accessed.

Example

The following example illustrates the use of the **arguments** property:

```
function ArgTest(){
   var i, s, numargs = arguments.length;
   s = numargs;
   if (numargs < 2)
      s += " argument was passed to ArgTest. It was ";
   else
      s += " arguments were passed to ArgTest. They were " ;
   for (i = 0; i < numargs; i++)
```

```
      {
         s += arguments[i] + " ";
      }
   return(s);
}
```

Requirements

Version 2

See Also

Arguments Object | function Statement

Applies To: Function Object

Build: Topic Version 5.6.9309.1546

JScript

callee Property

Returns the **Function** object being executed, that is, the body text of the specified **Function** object.

[*function*.]arguments.callee

The optional *function* argument is the name of the currently executing **Function** object.

Remarks

The **callee** property is a member of the **arguments** object that becomes available only when the associated function is executing.

The initial value of the **callee** property is the **Function** object being executed. This allows anonymous functions to be recursive.

Example

```
function factorial(n){
  if (n <= 0)
  return 1;
  else
    return n * arguments.callee(n - 1)
}
print(factorial(3));
```

Requirements

Version 5.5

See Also

Applies To: arguments Object | Function object

Build: Topic Version 5.6.9309.1546

JScript

caller Property

Returns a reference to the function that invoked the current function.

functionName.**caller**

The *functionName* object is the name of any executing function.

Remarks

The **caller** property is only defined for a function while that function is executing. If the function is called from the top level of a JScript program, **caller** contains **null**.

If the **caller** property is used in a string context, the result is the same as *functionName*.**toString**, that is, the decompiled text of the function is displayed.

The following example illustrates the use of the **caller** property:

```
function CallLevel(){
   if (CallLevel.caller == null)
      return("CallLevel was called from the top level.");
   else
      return("CallLevel was called by another function.");
}
```

Requirements

Version 2

See Also

function Statement

Applies To: Function Object

Build: Topic Version 5.6.9309.1546

JScript

constructor Property

Specifies the function that creates an object.

object.**constructor**

The required *object* 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:

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

or

```
function MyFunc {
   // Body of function.
}

y = new MyFunc;
if (y.constructor == MyFunc)
   // Do something (the condition will be true).
```

Requirements

Version 2

See Also

prototype Property

Applies To: Array Object | Boolean Object | Date Object | Function Object | Math Object | Number Object | Object Object | String Object

Build: Topic Version 5.6.9309.1546

JScript

description Property

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

object.**description** [**=** *stringExpression*]

Arguments

object
Required. Any instance of an **Error** object.
stringExpression
Optional. A string expression containing a description of the error.

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:

```
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 undefined".
}
```

Requirements

Version 5

See Also

number Property | message Property | name Property

Applies To: Error Object

Build: Topic Version 5.6.9309.1546

JScript

E Property

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

```
numVar = Math.E
```

Requirements

Version 1

See Also

exp Method | Math Object Properties

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

global Property

Returns a Boolean value indicating the state of the global flag (**g**) used with a regular expression. Default is **false**. Read-only.

rgExp.**global**

The required *rgExp* reference is an instance of a **Regular Expression** object.

Remarks

The **global** property returns **true** if the global flag is set for a regular expression, and returns **false** if it is not.

The global flag, when used, indicates that a search should find all occurrences of the pattern within the searched string, not just the first one. This is also known as global matching.

Example

The following example illustrates the use of the **global** property. If you pass "g" in to the function shown below, all instances of the word "the" are replaced with the word "a". Note that "The" at the beginning of the string is not replaced. This is because the initial letter is uppercase and, therefore, does not match the lowercase "t" in "the".

This function returns a string with a table that shows the condition of the properties associated with the allowable regular expression flags, **g**, **i**, and **m**. The function also returns the string with all replacements made.

```
function RegExpPropDemo(flag){
   if (flag.match(/[^gim]/))        //Check flag for validity.
     return("Flag specified is not valid");
```

```
   var r, re, s                    //Declare variables.
   var ss = "The man hit the ball with the bat.\n";
   ss += "while the fielder caught the ball with the glove.";
   re = new RegExp("the",flag);    //Specify the pattern to search for.
   r = ss.replace(re, "a");        //Replace "the" with "a".
   s = "Regular Expression property values:\n\n"
   s += "global  ignoreCase  multiline\n"
   if (re.global)                  //Test for global flag.
     s += " True     ";
   else
     s += "False     ";
   if (re.ignoreCase)              //Test ignoreCase flag.
     s += " True  ";
   else
     s += "False  ";
   if (re.multiline)               //Test multiline flag.
     s += "     True     ";
   else
     s += "     False   ";
   s += "\n\nThe resulting string is:\n\n" + r;
   return(s);                      //Returns replacement string
}
```

Requirements

Version 5.5

See Also

ignoreCase Property | multiline Property | Regular Expression Syntax

Applies To: RegExp Object

Build: Topic Version 5.6.9309.1546

JScript

hasOwnProperty Method

Returns a Boolean value indicating whether an object has a property with the specified name.

object.**hasOwnProperty(***proName***)**

Arguments

object
Required. Instance of an object.
proName
Required. String value of a property name.

Remarks

The **hasOwnProperty** method returns **true** if *object* has a property of the specified name, **false** if it does not. This method does not check if the property exists in the object's prototype chain; the property must be a member of the object itself.

Example

In the following example, all **String** objects share a common split method. The following code will print **false** and **true**.

```
var s = new String("JScript");
print(s.hasOwnProperty("split"));
print(String.prototype.hasOwnProperty("split"));
```

Requirements

Version 5.5

See Also

in Operator

Applies To: Object Object

Build: Topic Version 5.6.9309.1546

JScript

ignoreCase Property

Returns a Boolean value indicating the state of the ignoreCase flag (**i**) used with a regular expression. Default is **false**. Read-only.

rgExp.**ignoreCase**

The required *rgExp* reference is an instance of the **RegExp** object.

Remarks

The **ignoreCase** property returns **true** if the ignoreCase flag is set for a regular expression, and returns **false** if it is not.

The ignoreCase flag, when used, indicates that a search should ignore case sensitivity when matching the pattern within the searched string.

Example

The following example illustrates the use of the **ignoreCase** property. If you pass "i" in to the function shown below, all instances of the word "the" are replaced with the word "a", including the initial "The". This is because with the ignoreCase flag set, the search ignores any case sensitivity. So "T" is the same as "t" for the purposes of matching.

This function returns a string with a table that shows the condition of the properties associated with the allowable regular expression flags, **g**, **i**, and **m**. The function also returns the string with all replacements made.

```
function RegExpPropDemo(flag){
   if (flag.match(/[^gim]/))        //Check flag for validity.
```

```
    return("Flag specified is not valid");
  var r, re, s                    //Declare variables.
  var ss = "The man hit the ball with the bat.\n";
  ss += "while the fielder caught the ball with the glove.";
  re = new RegExp("the",flag);    //Specify the pattern to search for.
  r = ss.replace(re, "a");        //Replace "the" with "a".
  s = "Regular Expression property values:\n\n"
  s += "global  ignoreCase  multiline\n"
  if (re.global)                  //Test for global flag.
    s += " True     ";
  else
    s += "False     ";
  if (re.ignoreCase)              //Test ignoreCase flag.
    s += " True  ";
  else
    s += "False  ";
  if (re.multiline)               //Test multiline flag.
    s += "     True     ";
  else
    s += "     False   ";
  s += "\n\nThe resulting string is:\n\n" + r;
  return(s);                      //Returns replacement string
}
```

Requirements

Version 5.5

See Also

global property | multiline Property | Regular Expression Syntax

Applies To: RegExp Object

Build: Topic Version 5.6.9309.1546

JScript

index Property

Returns the character position where the first successful match begins in a searched string. Read-only.

RegExp.index

The object associated with this property is always the global **RegExp** object.

Remarks

The **index** property is zero-based. The initial value of the **index** property is –1. Its value changes whenever a successful match is made.

Example

The following example illustrates the use of the **index** property. This function iterates a search string and prints out the **index** and **lastIndex** values for each word in the string.

```
function RegExpTest(){
  var ver = Number(ScriptEngineMajorVersion() + "." + ScriptEngineMinorVersion())
  if (ver >= 5.5){
    var src = "The rain in Spain falls mainly in the plain.";
    var re = /\w+/g;
    var arr;
    while ((arr = re.exec(src)) != null)
       print(arr.index + "-" + arr.lastIndex + "\t" + arr);
  }
  else{
    alert("You need a newer version of JScript for this to work");
  }
}
```

Requirements

Version 3

See Also

RegExp Object Properties | Regular Expression Syntax

Applies To: RegExp Object

Build: Topic Version 5.6.9309.1546

JScript

Infinity Property

Returns an initial value of **Number.POSITIVE_INFINITY**.

```
Infinity
```

Remarks

The **Infinity** property is a member of the **Global** object, and is made available when the scripting engine is initialized.

Requirements

Version 3

See Also

POSITIVE_INFINITY Property | NEGATIVE_INFINITY Property

Applies To: Global Object

Build: Topic Version 5.6.9309.1546

JScript

input Property ($_)

Returns the string against which a regular expression search was performed. Read-only.

RegExp.input

The object associated with this property is always the global **RegExp** object.

Remarks

The value of **input** property is modified any time the searched string is changed.

The following example illustrates the use of the **input** property:

```
function inputDemo(){
   var s;
   var re = new RegExp("d(b+)(d)","ig");
   var str = "cdbBdbsbddbdz";
   var arr = re.exec(str);
   s = "The string used for the match was " + RegExp.input;
   return(s);
}
```

Requirements

Version 3

See Also

RegExp Object Properties | Regular Expression Syntax

Applies To: RegExp Object

Build: Topic Version 5.6.9309.1546

JScript

isPrototypeOf Method

Returns a Boolean value indicating whether an object exists in another object's prototype chain.

object1.**isPrototypeOf(***object2***)**

Arguments

object1
Required. Instance of an object.
object2
Required. Another object whose protoype chain is to be checked.

Remarks

The **isPrototypeOf** method returns **true** if *object2* has *object1* in its prototype chain. The prototype chain is used to share functionality between instances of the same object type. The **isPrototypeOf** method returns **false** when *object2* is not an object or when *object1* does not appear in the prototype chain of the *object2*.

Example

The following example illustrates the use of the **isPrototypeof** method.

```
function test(){
  var re = new RegExp();                        //Initialize variable.
```

```
  return (RegExp.prototype.isPrototypeOf(re));  //Return true.
}
```

Requirements

Version 5.5

See Also

Applies To: Object Object

Build: Topic Version 5.6.9309.1546

JScript

lastIndex Property

Returns the character position where the next match begins in a searched string.

RegExp.lastIndex

The object associated with this property is always the global **RegExp** object.

Remarks

The **lastIndex** property is zero-based, that is, the index of the first character is zero. It's initial value is –1. Its value is modified whenever a successful match is made.

The **lastIndex** property is modified by the **exec** and **test** methods of the **RegExp** object, and the **match**, **replace**, and **split** methods of the **String** object.

The following rules apply to values of **lastIndex**:

- If there is no match, **lastIndex** is set to -1.
- If **lastIndex** is greater than the length of the string, **test** and **exec** fail and **lastIndex** is set to -1.
- 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 -1.
- Otherwise, **lastIndex** is set to the next position following the most recent match.

Example

The following example illustrates the use of the **lastIndex** property. This function iterates a search string and prints out the **index** and **lastIndex** values for each word in the string.

```
function RegExpTest(){
  var ver = Number(ScriptEngineMajorVersion() + "." + ScriptEngineMinorVersion())
  if (ver >= 5.5){
    var src = "The rain in Spain falls mainly in the plain.";
    var re = /\w+/g;
    var arr;
    while ((arr = re.exec(src)) != null)
       print(arr.index + "-" + arr.lastIndex + "\t" + arr);
  }
  else{
    alert("You need a newer version of JScript for this to work");
  }
}
```

Requirements

Version 3

See Also

RegExp Object Properties | Regular Expression Syntax

Applies To: RegExp Object

Build: Topic Version 5.6.9309.1546

JScript

leftContext Property ($`)

Returns the characters from the beginning of a searched string up to the position before the beginning of the last match. Read-only.

RegExp.leftContext

The object associated with this property is always the global **RegExp** object.

Remarks

The initial value of the **leftContext** property is an empty string. The value of the **leftContext** property changes whenever a successful match is made.

Example

The following example illustrates the use of the **leftContext** property:

```
function matchDemo(){
   var s;                              //Declare variable.
   var re = new RegExp("d(b+)(d)","ig"); //Regular expression pattern.
   var str = "cdbBdbsbdbdz";           //String to be searched.
   var arr = re.exec(str);             //Perform the search.
   s = "$1 returns: " + RegExp.$1 + "\n";
   s += "$2 returns: " + RegExp.$2 + "\n";
   s += "$3 returns: " + RegExp.$3 + "\n";
   s += "input returns : " + RegExp.input + "\n";
   s += "lastMatch returns: " + RegExp.lastMatch + "\n";
   s += "leftContext returns: " + RegExp.leftContext + "\n";
   s += "rightContext returns: " + RegExp.rightContext + "\n";
   s += "lastParen returns: " + RegExp.lastParen + "\n";
   return(s);                          //Return results.
}
```

```
document.write(matchDemo());
```

Requirements

Version 5.5

See Also

$1...$9 Properties | index Property | input Property | lastIndex Property | lastMatch Property | lastParen Property | rightContext Property

Applies To: RegExp Object

Build: Topic Version 5.6.9309.1546

JScript

length Property (arguments)

Returns the actual number of arguments passed to a function by the caller.

[*function*.]**arguments.length**

The optional *function* argument is the name of the currently executing **Function** object.

Remarks

The **length** property of the **arguments** object is initialized by the scripting engine to the actual number of arguments passed to a **Function** object when execution begins in that function.

Example

The following example illustrates the use of the **length** property of the **arguments** object. To fully understand the example, pass more arguments to the function than the 2 arguments expected:

```
function ArgTest(a, b){
   var i, s = "The ArgTest function expected ";
   var numargs = 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);
}
```

Requirements

Version 5.5

See Also

arguments Property | length Property (Array) | length Property (String)

Applies To: arguments Object | Function object

Build: Topic Version 5.6.9309.1546

JScript

length Property (Array)

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

numVar = *arrayObj*.**length**

Arguments

numVar
Required. Any numeric variable.
arrayObj
Required. Any **Array** object.

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, `my_array.length` contains 7, not 2:

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

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:

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

Requirements

Version 2

See Also

length Property (Function) | length Property (String)

Applies To: Array Object

Build: Topic Version 5.6.9309.1546

JScript

lastMatch Property ($&)

Returns the last matched characters from any regular expression search. Read-only.

RegExp.lastMatch

The object associated with this property is always the global **RegExp** object.

Remarks

The initial value of the **lastMatch** property is an empty string. The value of the **lastMatch** property changes whenever a successful match is made.

Example

The following example illustrates the use of the **lastMatch** property:

```
function matchDemo(){
   var s;                              //Declare variable.
   var re = new RegExp("d(b+)(d)","ig"); //Regular expression pattern.
   var str = "cdbBdbsbdbdz";           //String to be searched.
```

```
   var arr = re.exec(str);              //Perform the search.
   s = "$1 returns: " + RegExp.$1 + "\n";
   s += "$2 returns: " + RegExp.$2 + "\n";
   s += "$3 returns: " + RegExp.$3 + "\n";
   s += "input returns : " + RegExp.input + "\n";
   s += "lastMatch returns: " + RegExp.lastMatch + "\n";
   s += "leftContext returns: " + RegExp.leftContext + "\n";
   s += "rightContext returns: " + RegExp.rightContext + "\n";
   s += "lastParen returns: " + RegExp.lastParen + "\n";
   return(s);                           //Return results.
}
document.write(matchDemo());
```

Requirements

Version 5.5

See Also

$1...$9 Properties | index Property | input Property | lastIndex Property | lastParen Property | leftContext Property | rightContext Property

Applies To: RegExp Object

Build: Topic Version 5.6.9309.1546

JScript

lastParen Property ($+)

Returns the last parenthesized submatch from any regular expression search, if any. Read-only.

RegExp.lastParen

The object associated with this property is always the global **RegExp** object.

Remarks

The initial value of the **lastParen** property is an empty string. The value of the **lastParen** property changes whenever a successful match is made.

Example

The following example illustrates the use of the **lastParen** property:

```
function matchDemo(){
   var s;                              //Declare variable.
   var re = new RegExp("d(b+)(d)","ig"); //Regular expression pattern.
   var str = "cdbBdbsbdbdz";           //String to be searched.
   var arr = re.exec(str);             //Perform the search.
   s = "$1 returns: " + RegExp.$1 + "\n";
   s += "$2 returns: " + RegExp.$2 + "\n";
   s += "$3 returns: " + RegExp.$3 + "\n";
   s += "input returns : " + RegExp.input + "\n";
   s += "lastMatch returns: " + RegExp.lastMatch + "\n";
   s += "leftContext returns: " + RegExp.leftContext + "\n";
   s += "rightContext returns: " + RegExp.rightContext + "\n";
   s += "lastParen returns: " + RegExp.lastParen + "\n";
   return(s);                          //Return results.
}
document.write(matchDemo());
```

Requirements

Version 5.5

See Also

$1...$9 Properties | index Property | input Property | lastIndex Property | lastMatch Property | leftContext Property | rightContext Property

Applies To: RegExp Object

Build: Topic Version 5.6.9309.1546

JScript

length Property (Function)

Returns the number of arguments defined for a function.

functionName.**length**

The required *functionName* is the name of the function.

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);
```

```
}
```

Requirements

Version 2

See Also

arguments Property | length Property (Array) | length Property (String)

Applies To: Function Object

Build: Topic Version 5.6.9309.1546

JScript

length Property (String)

Returns the length of a **String** object.

```
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.

Requirements

Version 1

See Also

length Property (Array) | length Property (Function) | String Object Methods | String Object Properties

Applies To: String Object

Build: Topic Version 5.6.9309.1546

JScript

LN10 Property

Returns the natural logarithm of 10.

numVar = **Math.LN10**

Remarks

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

Requirements

Version 1

See Also

Math Object Properties

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

LN2 Property

Returns the natural logarithm of 2.

numVar = **Math.LN2**

Syntax

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

Requirements

Version 1

See Also

Math Object Properties

Applies To: Math Object

© 2001 Microsoft Corporation. All rights reserved.

Build: Topic Version 5.6.9309.1546

JScript

LOG10E Property

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

varName = **Math.LOG10E**

Remarks

The **LOG10E** property, a constant, is approximately equal to 0.434.

Requirements

Version 1

See Also

Math Object Properties

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

LOG2E Property

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

varName = **Math.LOG2E**

Remarks

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

Requirements

Version 1

See Also

Math Object Properties

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

MAX_VALUE Property

Returns the largest number representable in JScript. Equal to approximately 1.79E+308.

Number.MAX_VALUE

The *number* argument is the **Number** object.

Remarks

The **Number** object does not have to be created before the **MAX_VALUE** property can be accessed.

Requirements

Version 2

See Also

MIN_VALUE Property | NaN Property | NEGATIVE_INFINITY Property | POSITIVE_INFINITY Property | toString Method

Applies To: Number Object

Build: Topic Version 5.6.9309.1546

JScript

message Property

Returns an error message string.

errorObj.**message**

Arguments

errorObj
Required. Instance of **Error** object.

Remarks

The **message** property is a string containing an error message displayed to users. It contains the same information as the **description** property.

Example

The following example causes a TypeError exception to be thrown, and displays the name of the error and its message.

```
try {
  // 'null' is not a valid object
  null.doSomething();
}
catch(e){
  print(e.name + ": " + e.message);
  print(e.number + ": " + e.description);
}
```

Requirements

Version 5

See Also

description Property | name Property

Applies To: Error Object

Build: Topic Version 5.6.9309.1546

JScript

MIN_VALUE Property

Returns the number closest to zero representable in JScript. Equal to approximately 5.00E-324.

Number.MIN_VALUE

The *number* argument is the **Number** object.

Remarks

The **Number** object does not have to be created before the **MIN_VALUE** property can be accessed.

Requirements

Version 2

See Also

MAX_VALUE Property | NaN Property | NEGATIVE_INFINITY Property | POSITIVE_INFINITY Property | toString Method

Applies To: Number Object

Build: Topic Version 5.6.9309.1546

JScript

multiline Property

Returns a Boolean value indicating the state of the multiline flag (**m**) used with a regular expression. Default is **false**. Read-only.

rgExp.**multiline**

The required *rgExp* argument is an instance of the **RegExp** object

Remarks

The **multiline** property returns **true** if the multiline flag is set for a regular expression, and returns **false** if it is not. The **multiline** property is **true** if the regular expression object was created with the **m** flag.

If **multiline** is **false**, "^" matches the position at the beginning of a string, and "$" matches the position at the end of a string. If **multline** is **true**, "^" matches the position at the beginning of a string as well as the position following a "\n" or "\r", and "$" matches the position at the end of a string and the position preceding "\n" or "\r".

Example

The following example illustrates the behavior of the **multiline** property. If you pass "m" in to the function shown below, the word "while" is replaced with the word "and". This is because with the multiline flag is set and the word "while" occurs at the beginning of the line after a newline character. The multiline flag allows the search to be performed on multiline strings.

This function returns a string with a table that shows the condition of the allowable regular expression flags, **g**, **i**, and **m**. The function also returns the string with all replacements made.

```
function RegExpPropDemo(flag){
   if (flag.match(/[^gim]/))          //Check flag for validity.
     return("Flag specified is not valid");
   var r, re, s                       //Declare variables.
   var ss = "The man hit the ball with the bat.";
   ss += "\nwhile the fielder caught the ball with the glove.";
   re = new RegExp("^while",flag);    //Specify the pattern to search for.
   r = ss.replace(re, "and");         //Replace "the" with "a".
   s = "Regular Expression property values:\n\n"
   s += "global  ignoreCase  multiline\n"
   if (re.global)                     //Test for global flag.
     s += " True     ";
   else
     s += "False     ";
   if (re.ignoreCase)                 //Test ignoreCase flag.
     s += " True  ";
   else
     s += "False  ";
   if (re.multiline)                  //Test multiline flag.
     s += "     True     ";
   else
     s += "     False   ";
   s += "\n\nThe resulting string is:\n\n" + r;
   return(s);                         //Returns replacement string
}
```

Requirements

Version 5.5

See Also

global property | ignoreCase Property | Regular Expression Syntax

Applies To: RegExp Object

Build: Topic Version 5.6.9309.1546

JScript

name Property

Returns the name of an error.

errorObj.**name**

Arguments

errorObj
Required. Instance of **Error** object.

Remarks

The **name** property returns the name or exception type of an error. When a runtime error occurs, the name property is set to one of the following native exception types:

Exception Type	Meaning
ConversionError	This error occurs whenever there is an attempt to convert an object into something to which it cannot be converted.
RangeError	This error occurs when a function is supplied with an argument that has exceeded its allowable range. For example, this error occurs if you attempt to construct an **Array** object with a length that is not a valid positive integer.
ReferenceError	This error occurs when an invalid reference has been detected. This error will occur, for example, if an expected reference is **null**.
RegExpError	This error occurs when a compilation error occurs with a regular expression. Once the regular expression is compiled, however, this error cannot occur. This example will occur, for example, when a regular expression is declared with a pattern that has an invalid syntax, or flags other than **i**, **g**, or **m**, or if it contains the same flag more than once.
SyntaxError	This error occurs when source text is parsed and that source text does not follow correct syntax. This error will occur, for example, if the **eval** function is called with an argument that is not valid program text.
TypeError	This error occurs whenever the actual type of an operand does not match the expected type. An example of when this error occurs is a function call made on something that is not an object or does not support the call.
URIError	This error occurs when an illegal Uniform Resource Indicator (URI) is detected. For example, this is error occurs when an illegal character is found in a string being encoded or decoded.

Example

The following example causes a TypeError exception to be thrown, and displays the name of the error and its message.

```
try {
  // 'null' is not a valid object
  null.doSomething();
}
catch(e){
  print(e.name + ": " + e.message);
  print(e.number + ": " + e.description);
}
```

Requirements

Version 5.5

See Also

description Property | message Property | number Property

Applies To: Error Object

Build: Topic Version 5.6.9309.1546

JScript

NaN Property

A special value that indicates an arithmetic expression returned a value that was not a number.

```
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.

Requirements

Version 2

See Also

isNaN Method | MAX_VALUE Property | MIN_VALUE Property | NEGATIVE_INFINITY Property | POSITIVE_INFINITY Property | toString Method

Applies To: Number Object

Build: Topic Version 5.6.9309.1546

JScript

NaN Property (Global)

Returns the special value **NaN** indicating that an expression is not a number.

```
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.

Requirements

Version 3

See Also

isNaN Method

Applies To: Global Object

Build: Topic Version 5.6.9309.1546

JScript

NEGATIVE_INFINITY Property

Returns a value more negative than the largest negative number (**-Number.MAX_VALUE**) representable in JScript.

```
Number.NEGATIVE_INFINITY
```

The *number* argument is the **Number** object.

Remarks

The **Number** object does not have to be created before the **NEGATIVE_INFINITY** property can be accessed.

JScript displays **NEGATIVE_INFINITY** values as -infinity. This value behaves mathematically as infinity.

Requirements

Version 2

See Also

MAX_VALUE Property | MIN_VALUE Property | NaN Property | POSITIVE_INFINITY Property | toString Method

Applies To: Number Object

Build: Topic Version 5.6.9309.1546

JScript

number Property

Returns or sets the numeric value associated with a specific error. The **Error** object's default property is **number**.

object.**number** [= *errorNumber*]

Arguments

object
Any instance of the **Error** object.
errorNumber
An integer representing an error.

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:

```
try
   x = y   // Cause an error.
catch(var e){   // Create local variable e.
   document.write(e)   // Prints "[object Error]".
   document.write(e.number>>16 & 0x1FFF)   // Prints 10, the facility code.
   document.write(e.number & 0xFFFF)   // Prints 5009, the error code.
   document.write(e.description)   // Prints "'y' is undefined".
}
```

Requirements

Version 5

See Also

description Property | message Property | name Property

Applies To: Error Object

Build: Topic Version 5.6.9309.1546

JScript

PI Property

Returns the ratio of the circumference of a circle to its diameter, approximately 3.141592653589793.

numVar = **Math.PI**

Syntax

The **PI** property, a constant, is approximately equal to 3.14159.

Requirements

Version 1

See Also

Math Object Properties

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

POSITIVE_INFINITY Property

Returns a value larger than the largest number (**Number.MAX_VALUE**) that can be represented in JScript.

```
Number.POSITIVE_INFINITY
```

The *number* argument is the **Number** object.

Remarks

The **Number** object does not have to be created before the **POSITIVE_INFINITY** property can be accessed.

JScript displays **POSITIVE_INFINITY** values as infinity. This value behaves mathematically as infinity.

Requirements

Version 2

See Also

MAX_VALUE Property | MIN_VALUE Property | NaN Property | NEGATIVE_INFINITY Property | toString Method

Applies To: Number Object

Build: Topic Version 5.6.9309.1546

JScript

propertyIsEnumerable Property

Returns a Boolean value indicating whether a specified property is part of an object and if it is enumerable.

object.**propertyIsEnumerable(***proName***)**

Arguments

object
Required. Instance of an object.
proName
Required. String value of a property name.

Remarks

The **propertyIsEnumerable** property returns **true** if *proName* exists in *object* and can be enumerated using a **For…In** loop. The **propertyIsEnumerable** property returns **false** if *object* does not have a property of the specified name or if the specified property is not enumerable. Typically, predefined properties are not enumerable while user-defined properties are always enumerable.

The **propertyIsEnumerable** property does not consider objects in the prototype chain.

Example

```
function testIsEnumerable(){
  var a = new Array("apple", "banana", "cactus");
  return(a.propertyIsEnumerable(1));
}
```

Requirements

Version 5.5

See Also

Applies To: Object Object

Build: Topic Version 5.6.9309.1546

JScript

prototype Property

Returns a reference to the prototype for a class of objects.

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.

```
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.

Requirements

Version 2

See Also

constructor Property

Applies To: Array Object | Boolean Object | Date Object | Function Object | Number Object | Object Object | String Object

---

Build: Topic Version 5.6.9309.1546

JScript

rightContext Property ($')

Returns the characters from the position following the last match to the end of the searched string. Read-only.

RegExp.rightContext

The object associated with this property is always the global **RegExp** object.

Remarks

The initial value of the **rightContext** property is an empty string. The value of the **rightContext** property changes whenever a successful match is made.

Example

The following example illustrates the use of the **rightContext** property:

```
function matchDemo(){
   var s;                              //Declare variable.
   var re = new RegExp("d(b+)(d)","ig"); //Regular expression pattern.
   var str = "cdbBdbsbddbdz";          //String to be searched.
   var arr = re.exec(str);             //Perform the search.
   s = "$1 returns: " + RegExp.$1 + "\n";
   s += "$2 returns: " + RegExp.$2 + "\n";
   s += "$3 returns: " + RegExp.$3 + "\n";
   s += "input returns : " + RegExp.input + "\n";
   s += "lastMatch returns: " + RegExp.lastMatch + "\n";
   s += "leftContext returns: " + RegExp.leftContext + "\n";
   s += "rightContext returns: " + RegExp.rightContext + "\n";
   s += "lastParen returns: " + RegExp.lastParen + "\n";
   return(s);                          //Return results.
}
document.write(matchDemo());
```

Requirements

Version 5.5

See Also

$1...$9 Properties | index Property | input Property | lastIndex Property | lastMatch Property | lastParen Property | leftContext Property

Applies To: RegExp Object

Build: Topic Version 5.6.9309.1546

JScript

source Property

Returns a copy of the text of the regular expression pattern. Read-only.

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:

```
function SourceDemo(re, s){
   var s1;
   // Test string for existence of regular expression.
   if (re.test(s))
      s1 = " contains ";
   else
      s1 = " does not contain ";
   // Get the text of the regular expression itself.
   return(s + s1 + re.source);
}
```

Requirements

Version 3

See Also

Regular Expression Object Methods | Regular Expression Object Properties | Regular Expression Syntax

Applies To: Regular Expression Object

Build: Topic Version 5.6.9309.1546

JScript

SQRT1_2 Property

Returns he square root of 0.5, or one divided by the square root of 2.

```
numVar = Math.SQRT1_2
```

Remarks

The **SQRT1_2** property, a constant, is approximately equal to 0.707.

Requirements

Version 1

See Also

Math Object Properties | sqrt Method | SQRT2 Property

Applies To: Math Object

Build: Topic Version 5.6.9309.1546

JScript

SQRT2 Property

Returns the square root of 2.

numVar = **Math.SQRT2**

Syntax

The **SQRT2** property, a constant, is approximately equal to 1.414.

Requirements

Version 1

See Also

Math Object Properties | sqrt Method | SQRT1_2 Property

Applies To: Math Object

© 2001 Microsoft Corporation. All rights reserved.

Build: Topic Version 5.6.9309.1546

JScript

undefined Property

Returns an initial value of **undefined**.

```
undefined
```

Remarks

The **undefined** property is a member of the **Global** object, and becomes available when the scripting engine is initialized. When a variable has been declared but not initialized, its value is **undefined**.

If a variable has not been declared, you cannot compare it to **undefined**, but you can compare the type of the variable to the string "undefined"

The **undefined** property is useful when explicitly testing or setting a variable to undefined.

Example

```
var declared;                          //Declare variable.
if (declared == undefined)             //Test variable.
  document.write("declared has not been given a value.");

if (typeOf(notDeclared) == "undefined")
  document.write("notDeclared has not been defined.");
```

Requirements

Version 5.5

See Also

Applies To: Global Object

Build: Topic Version 5.6.9309.1546

JScript

JScript Statements

The following table lists JScript statements.

Description	Language Element
Terminates the current loop, or if in conjunction with a *label*, terminates the associated statement.	break Statement
Contains statements to execute when an error occurs in code within the **try** block.	catch Statement
Activates conditional compilation support.	@cc_on Statement
Causes single-line comments to be ignored by the JScript parser.	// (Single-line Comment Statement)
Causes multiline comments to be ignored by the JScript parser.	/*..*/ (Multiline Comment Statement)
Stops the current iteration of a loop, and starts a new iteration.	continue Statement
Executes a statement block once, and then repeats execution of the loop until a condition expression evaluates to **false**.	do...while Statement
Executes a block of statements for as long as a specified condition is **true**.	for Statement
Executes one or more statements for each element of an object or array.	for...in Statement
Declares a new function.	function Statement
Conditionally executes a group of statements, depending on the value of an expression.	@if Statement
Conditionally executes a group of statements, depending on the value of an expression.	if...else Statement
Provides an identifier for a statement.	Labeled Statement
Exits from the current function and returns a value from that function.	return Statement
Creates variables used with conditional compilation statements.	@set Statement
Enables the execution of one or more statements when a specified expression's value matches a label.	switch Statement
Refers to the current object.	this Statement

Generates an error condition that can be handled by a **try...catch** statement.	throw Statement
Implements error handling for JScript.	try Statement
Declares a variable.	var Statement
Executes a statement until a specified condition is **false**.	while Statement
Establishes the default object for a statement.	with Statement

---

Build: Topic Version 5.6.9309.1546

JScript

@cc_on Statement

Activates conditional compilation support.

@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.

Requirements

Version 3

See Also

Conditional Compilation | Conditional Compilation Variables | @if Statement | @set Statement

---

Build: Topic Version 5.6.9309.1546

JScript

@if Statement

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

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

Arguments

condition1, *condition2*
Optional. An expression that can be coerced into a Boolean expression.

text1
Optional. Text to be parsed if *condition1* is **true**.

text2
Optional. Text to be parsed if *condition1* is **false** and *condition2* is **true**.
text3
Optional. Text to be parsed if both *condition1* and *condition2* are **false**.

Remarks

When you write an **@if** statement, you do not 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:

```
alert(@if (@_win32) "using Windows NT or Windows 95" @else "using Windows 3.1" @end)
```

Requirements

Version 3

See Also

Conditional Compilation | Conditional Compilation Variables | @cc_on Statement | @set Statement

---

Build: Topic Version 5.6.9309.1546

JScript

@set Statement

Creates variables used with conditional compilation statements.

```
@set @varname = term
```

Arguments

varname
Required. Valid JScript variable name. Must be preceded by an "@" character at all times.

term
Required. Zero or more unary operators followed by a constant, conditional compilation variable, or parenthesized expression.

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:

```
@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.

Requirements

Version 3

See Also

Conditional Compilation | Conditional Compilation Variables | @cc_on Statement | @if Statement

---

Build: Topic Version 5.6.9309.1546

JScript

break Statement

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

```
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.

Example

The following example illustrates the use of the **break** statement.

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

Requirements

Version 1

See Also

continue Statement | do...while Statement | for Statement | for...in Statement | Labeled Statement | while Statement

---

Build: Topic Version 5.6.9309.1546

JScript

Comment Statements

Causes comments to be ignored by the JScript parser.

Syntax 1

```
Single-line Comment:
```

```
// comment
```

Syntax 2

```
Multiline Comment:
/*
comment
*/
```

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

Syntax 3

```
//@CondStatement
```

Syntax 4

```
/*@
condStatement
@*/
```

The c*ondStatement* 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.

Example

The following example illustrates the most common uses of the **comment** statement.

```
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);
}
```

Requirements

Version 1

Build: Topic Version 5.6.9309.1546

JScript

continue Statement

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

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.

Example

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);
}
```

Requirements

Version 1

See Also

break Statement | do...while Statement | for Statement | for...in Statement | Labeled Statement | while Statement

---

Build: Topic Version 5.6.9309.1546

JScript

do...while Statement

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

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

Arguments

statement
Optional. The statement to be executed if *expression* is **true**. Can be a compound statement.

expression
Optional. An expression that can be coerced to Boolean **true** or **false**. If *expression* is **true**, the loop is executed again. If *expression* is **false**, the loop is terminated.

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.

Example

The following example illustrates the use of the **do...while** statement to iterate the **Drives** collection.

```
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);
}
```

Requirements

Version 3

See Also

break Statement | continue Statement | for Statement | for...in Statement | while Statement | Labeled Statement

---

Build: Topic Version 5.6.9309.1546

JScript

for Statement

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

```
for (initialization; test; increment)
   statements
```

Arguments

initialization

Required. An expression. This expression is executed only once, before the loop is executed.

test

Required. A Boolean expression. If *test* is **true**, *statement* is executed. If *test* if **false**, the loop is terminated.

increment

Required. An expression. The increment expression is executed at the end of every pass through the loop.

statements

Optional. One or more statements to be executed if *test* is **true**. Can be a compound statement.

Remarks

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

Example

The following example demonstrates a **for** loop.

```
/* 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;
}
```

Requirements

Version 1

See Also

for...in Statement | while Statement

Build: Topic Version 5.6.9309.1546

JScript

for...in Statement

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

```
for (variable in [object | array])
   statements
```

Arguments

variable
Required. A variable that can be any property of *object* or any element of an *array*.
object, array
Optional. An object or array over which to iterate.
statements
Optional. One or more statements to be executed for each property of *object* or each element of *array*. Can be a compound statement.

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, ...

Example

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

```
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] + "&ltBR>";
   }
   return(s);
}
```

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

Requirements

Version 5

See Also

for Statement | while Statement

Build: Topic Version 5.6.9309.1546

JScript

function Statement

Declares a new function.

```
function functionname([arg1 [, arg2 [,...[, argN]]]])
{
   statements
}
```

Arguments

functionname
Required. The name of the function.

arg1...argN
Optional. An optional, comma-separated list of arguments the function understands.
statements
Optional. One or more JScript statements.

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.

Example

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

```
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.

Requirements

Version 1

See Also

new Operator

Build: Topic Version 5.6.9309.1546

JScript

if...else Statement

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

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

Arguments

condition
Required. A Boolean expression. If *condition* is null or undefined, *condition* is treated as **false**.

statement1
Optional. The statement to be executed if *condition* is **true**. Can be a compound statement.

statement2
Optional. The statement to be executed if *condition* is **false**. Can be a compound statement.

Remarks

It is generally good practice to enclose *statement1* and *statement2* in braces ({}) for clarity and to avoid inadvertent errors.

Example

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 not use braces, you can accidentally create an error:

```
if (x == 5)
   z = 7;
   q = 42;
else
   z = 19;
```

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

Requirements

Version 1

See Also

Conditional Operator (?:)

Build: Topic Version 5.6.9309.1546

JScript

Labeled Statement

Provides an identifier for a statement.

```
label :
   statements
```

Arguments

label
Required. A unique identifier used when referring to the labeled statement.
statements
Optional. One or more statements associated with *label*.

Remarks

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

Example

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];
         }
         s1 += s + "\n";
      }
      return(s1)
}
```

Requirements

Version 3

See Also

break Statement | continue Statement

Build: Topic Version 5.6.9309.1546

JScript

return Statement

Exits from the current function and returns a value from that function.

return[(][*expression*][)];

The optional *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.

Example

The following example illustrates the use of the **return** statement.

```
function myfunction(arg1, arg2){
   var r;
   r = arg1 * arg2;
   return(r);
}
```

Requirements

Version 1

See Also

function Statement

Build: Topic Version 5.6.9309.1546

JScript

switch Statement

Enables the execution of one or more statements when a specified expression's value matches a label.

```
switch (expression) {
   case label :
       statementlist
   case label :
       statementlist
   ...
```

```
   default :
       statementlist
}
```

Arguments

expression
The expression to be evaluated.

label
An identifier to be matched against *expression*. If *label* === *expression*, execution starts with the *statementlist* immediately after the colon, and continues until it encounters either a **break** statement, which is optional, or the end of the **switch** statement.

statementlist
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:

- Evaluate *expression* and look at *label* in order until a match is found.
- 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.
- If no *label* equals *expression*, go to the **default** case. If there is no **default** case, go to last step.
- Continue execution at the statement following the end of the **switch** code block.

Example

The following example tests an object for its type.

```
function MyObject() {
...}
```

```
switch (object.constructor){
   case Date:
   ...
   case Number:
   ...
   case String:
   ...
   case MyObject:
   ...
   default:
   ...
}
```

Requirements

Version 3

See Also

break Statement | if...else Statement

Build: Topic Version 5.6.9309.1546

JScript

this Statement

Refers to the current object.

this.*property*

The required *property* argument is one of the current object's properties

Remarks

The **this** keyword is typically used in object constructors to refer to the current object.

Example

In the following example, **this** refers to the newly created Car object, and assigns values to three properties:

```
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.

Requirements

Version 1

See Also

new Operator

Build: Topic Version 5.6.9309.1546

JScript

throw Statement

Generates an error condition that can be handled by a **try...catch…finally** statement.

throw *exception*

The required *exception* argument can be any expression.

Remarks

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

```
function TryCatchDemo(x){
   try {
      try {
      if (x == 0)   // Evalute argument.
         throw "x equals zero";   // Throw an error.
      else
         throw "x does not equal zero";   // Throw a different error.
      }
      catch(e) {   // Handle "x = 0" errors here.
         if (e == "x equals zero")   // Check for an error handled here.
            return(e + " handled locally.");   // Return object error message.
         else   // Can't handle error here.
            throw e;   // Rethrow the error for next
      }   // error handler.
   }
   catch(e) {   // Handle other errors here.
      return(e + " handled higher up.");   // Return error message.
   }
}
document.write(TryCatchDemo(0));
document.write(TryCatchDemo(1));
```

Requirements

Version 5

See Also

try...catch Statement

Build: Topic Version 5.6.9309.1546

JScript

try...catch…finally Statement

Implements error handling for JScript.

```
try {
   tryStatements}
catch(exception){
   catchStatements}
finally {
   finallyStatements}
```

Arguments

tryStatements
Required. Statements where an error can occur.
exception
Required. Any variable name. The initial value of *exception* is the value of the thrown error.
catchStatements
Optional. Statements to handle errors occurring in the associated *tryStatements*.
finallyStatements
Optional. Statements that are unconditionally executed after all other error processing has occurred.

Remarks

The **try...catch…finally** 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 *tryStatements* contain code where an error can occur, while *catchStatements* contain the code to handle any error that does occur. If an error occurs in the *tryStatements*, program control is passed to *catchStatements* for processing. The initial value of *exception* is the value of the error that occurred in *tryStatements*. If no error occurs, *catchStatements* are never executed.

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

After all statements in *tryStatements* have been executed and any error handling has occurred in *catchStatements*, the statements in *finallyStatements* are unconditionally executed.

Notice that the code inside *finallyStatements* is executed even if a return statement occurs inside the **try** or **catch** blocks, or if the **catch** block re-throws the error. *finallyStatments* are guaranteed to always run, unless an unhandled error occurs (for example, causing a run-time error inside the **catch** block).

Example

The following example shows how JScript exception handling works.

```
try {
  print("Outer try running..");
  try {
    print("Nested try running...");
    throw "an error";
  }
  catch(e) {
    print("Nested catch caught " + e);
    throw e + " re-thrown";
  }
  finally {
    print("Nested finally is running...");
  }
}
catch(e) {
  print("Outer catch caught " + e);
}
finally {
  print("Outer finally running");
}
// Change this for Windows Script Host to say WScript.Echo(s)
function print(s){
```

```
   document.write(s);
}
```

This produces the following output:

```
Outer try running..
Nested try running...
Nested catch caught an error
Nested finally is running...
Outer catch caught an error re-thrown
Outer finally running
```

Requirements

Version 5

See Also

throw Statement

Build: Topic Version 5.6.9309.1546

JScript

var Statement

Declares a variable.

var *variable1* [= *value1*] [, *variable2* [= *value2*], ...]

Arguments

variable1, variable2
The names of the variables being declared.
value1, value2
The initial value assigned to the variable.

Remarks

Use the **var** statement to declare variables. These variables can be assigned values at declaration or later in your script.

Example

The following example illustrates the use of the **var** statement.

```
var index;
var name = "Thomas Jefferson";
var answer = 42, counter, numpages = 10;
```

Requirements

Version 1

See Also

function Statement | new Operator

Build: Topic Version 5.6.9309.1546

JScript

while Statement

Executes a statement until a specified condition is **false**.

```
while (expression)
   statements
```

Arguments

expression
: Required. A Boolean expression checked before each iteration of the loop. If *expression* is **true**, the loop is executed. If *expression* is **false**, the loop is terminated.

statements
: Optional. One or more statements to be executed if *expression* is **true**.

Remarks

The **while** statement checks *expression* before a loop is first executed. If *expression* is **false** at this time, the loop is never executed.

Example

The following example illustrates the use of the **while** statement.

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

Requirements

Version 1

See Also

break Statement | continue Statement | do...while Statement | for Statement | for...in Statement

---

Build: Topic Version 5.6.9309.1546

JScript

with Statement

Establishes the default object for a statement.

```
with (object)
   statements
```

Arguments

object
The new default object.
statements
One or more statements for which *object* is the default object.

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:

```
with (Math){
```

```
   x = cos(3 * PI) + sin (LN10)
   y = tan(14 * E)
}
```

Requirements

Version 1

See Also

this Statement

Build: Topic Version 5.6.9309.1546