Quick Reference Library

JavaScript Quick Reference

Welcome to the **DevGuru JavaScript Quick Reference** guide. This is an extensive 214 page reference source that explains and gives comprehensive, working examples of code in a definitive manner for the JavaScript language (and hence, for the ECMAScript and JScript languages). All elements of the language are covered, including the events, functions, methods, objects, operators, properties, statements, and values.

The JavaScript language was developed by the Netscape Communications Corporation and is a trademarked name. It is a cross-platform, object-based scripting language that was originally designed for use in Netscape Navigator. Indeed, versions 2.0, and later, of Navigator can interpret JavaScript statements that are embedded within HTML code. When a request is made to see a page, the HTML code that defines the requested page along with the embedded JavaScript statements, are sent by the server to the client. Navigator interprets the HTML document and executes the JavaScript code. The resultant page is displayed for the client. It is important to understand that this interpretation occurs on the client-side rather than the server-side.

After the success of JavaScript in Navigator 2.0, the Microsoft Corporation was quick to create a clone of JavaScript, called JScript, which is a trademarked name, that is designed to run inside the Microsoft Internet Explorer. In truth, except for a few minor differences, JScript is essentially a carbon copy of JavaScript. As a consequence, this **DevGuru JavaScript Quick Reference** will be of value to both JavaScript and JScript users.

The latest versions of JavaScript and JScript are compliant with the European Computer Manufacturing Association's ECMAScript Language Specification (ECMA-262 standard, for short). Note that the name for this ECMA-262 language is ECMAScript. However, Netscape will continue to use the name, JavaScript and, likewise, Microsoft will continue to use the name, JScript. It is important to understand that the ECMA-262 standards sets minimum compatibility requirements. You should expect current and future versions of both JavaScript and JScript to also contain additional proprietary features, beyond the minimum requirements, designed to woo the developer to favor one language over the other. Fortunately, both Microsoft and Netscape have promised to submit new features to ECMA for inclusion in the evolving ECMA-262 standard. Many older browsers are, of course, still very happily utilizing older, non-compliant versions of these scripting languages.

JavaScript is a simple to comprehend, easy to use, general purpose scripting language. When used in conjunction with a Web browser's Document Object Model (DOM), it can produce powerful dynamic HTML browser-based applications which also can feature animation and sound.

INDEX

OPERATORS: + ++ - -- * / %

JavaScript contains the standard arithmetic operators plus ones for modulus, increment, decrement and unary negation.

+

This is the standard addition operator and returns the sum of two numerical values (either literals or variables):

Code:
```
x = a + 6;
```

++

This is the increment operator and is used to increment (add one to) its operand. The position of the operator in relation to its operand determines whether the increment takes place before or after returning a value. If it is placed after the operand, then it returns the value before incrementing as in the following example which, assuming 'a' to be 5, sets 'x' to 5 and then increments 'a' to 6:

Code:
```
x = a++;
```

...whereas, if the operator is placed before the operand, 'a' of the above example will first be incremented to 6, and then the value 6 will be assigned to 'x':

Code:
```
x = ++a;
```

-

This is the standard subtraction operator and it subtracts one number from another:

Code:
```
x = a - 6;
```

This is also the unary negation operator which precedes and negates its operand. In the following example with 'a' being 6, the value -6 is assigned to the variable 'x' while 'a' retains the value 6:

Code:
```
x = -a;
```

--

This is the decrement operator which decrements (deducts one from) its operand. As with the increment operator, the position of the decrement operator in relation to its operand determines whether the decrement takes place before or after the assignment operation. If it is placed after the operand, then a value is returned before decrementing, as in the next example which, assuming 'a' to be 6, assigns 6 to the variable 'x' and then decrements 'a' to 5:

Code:

```
x = a--;
```

...while, if the decrement operator is placed before its operand, the decrement takes place before the assignment. Assuming 'a' to be 6, this next example decrements 'a' to 5 and then sets the variable 'x' to 5:

Code:
```
x = --a;
```

***

This is the standard multiplication operator and it returns the product of two numerical values (either literals or variables):

Code:
```
x = a * 7;
```

/ This is the standard division operator which divides one number by another:

Code:
```
x = a / 7;
```

%

This is the modulus operator which returns the integer remainder of dividing the preceding operand by the one following it. The next example returns the value 2 to the variable 'x':

Code:
```
x = 9 % 7;
```

Operator: \ (backslash inline or escaped characters)

escape(string)

The \ (backslash) is used to insert apostrophes, carriage returns, quotes, tabs, and other special characters inside a string.

For example, in JavaScript, the start and stop of a string is delimited with either single or double quotes. However, if the string contains single and/or double quotes, you have problems.

Consider the string, "My favorite rose is the "Peach Delight.""

JavasScript will chop the output to, My favorite rose is the

Fortunately, this has a very simple solution. All you have to do is to place a backslash \ before each double quote in "Peach Delight". This turns each \" into a string literal.

The string is becomes, "My favorite rose is the \"Peach Delight\"."

JavasScript will now output, My favorite rose is the "Peach Delight."

This same concept is used with a variety of other characters. These backslash pairs are refered to as inline or escaped characters.

Code:

\'	single quote
\"	double quote
\\\\	backslash
\b	backspace
\f	form feed
\n	new line
\r	carriage return
\t	tab

OPERATORS: =

The basic assignment operator is the equal sign, which assigns the value (literal or variable) on its right to the variable on its left:

Code:

```
x = a;
```

The assignment operator can also be combined with a variety of other operators to provide shorthand versions of standard operations. It combines with the arithmetic operators **+**, **-**, *****, **/** and **%** to give the following shorthand versions:

```
a += b    instead of   a = a + b
a -= b    instead of   a = a - b
a *= b    instead of   a = a * b
a /= b    instead of   a = a / b
a %= b    instead of   a = a % b
```

It also combines with the following bitwise operators:

```
a <<= b    instead of   a = a << b
a >>= b    instead of   a = a >> b
a >>>= b    instead of   a = a >>> b
a &= b    instead of   a = a & b
a ^= b    instead of   a = a ^ b
a |= b    instead of   a = a | b
```

NOTE:

If the operand to the left of the equal sign itself contains an assignment operator, then the above shorthand statements won't work. Instead you must use the longer version of the statement.

OPERATORS: & | ^ ~ << >> >>>

Bitwise operators perform logical and shift operations. They work by treating their operands as a series of 32 bits and performing their operations on them at this bit level. However, the return value is a decimal format number.

The following examples of bitwise logical operators assume the variable 'a' to be 13 (binary 1101) and 'b' to be 9 (binary 1001)

&

This is the bitwise AND operator which returns a 1 for each bit position where the corresponding bits of both its operands are 1. The following code would return 9 (1001):

Code:
```
result = a & b;
```

|

This is the bitwise OR operator and returns a one for each bit position where one or both of the corresponding bits of its operands is a one. This example would return 13 (1101):

Code:
```
result = a | b;
```

^ This is the bitwise XOR operator, which returns a one for each position where one (not both) of the corresponding bits of its operands is a one. The next example returns 4 (0100):

Code:
```
result = a ^ b;
```

~

This is the bitwise NOT operator and it works by converting each bit of its operand to its opposite. This example returns -14:

Code:
```
result = ~a;
```

The following bitwise operators perform shift operations. In the examples the variable 'a' is assumed to be 13 (binary 1101) and the variable 'b' 2 (binary 10).

<<

This is the left shift operator and it works by shifting the digits of the binary representation of the first operand to the left by the number of places specified by the second operand. The spaces created to the right are filled in by zeros, and any digits falling off the left are discarded. The following code returns 52 as the binary of 13 (1101) is shifted two places to the left giving 110100:

Code:
```
result = a << b;
```

>>

This is the sign-propagating right shift operator which shifts the digits of the binary representation of the first operand to the right by the number of places specified by the second operand, discarding any shifted off to the right. The copies of the leftmost bit are added on from the left, thereby preserving the sign of the number. This next example returns 3 (11) as the two right-most bit of 13 (1101) are shifted off to the right and discarded:

Code:
```
result = a >> b;
```

Note that if 'a' were -13 in the above example, the code would return -4 as the sign is preserved.

>>>

This is the zero-fill right shift operator which shifts the binary representation of the first operand to the right by the number of places specified by the second operand. Bits shifted off to the right are discarded and zeroes are added on to the left. With a positive number you would get the same result as with the sign-propagating right shift operator, but negative numbers lose their sign becoming positive as in the next example, which (assuming 'a' to be -13) would return 1073741820:

Code:
```
result = a >>> b;
```

OPERATORS: == != === !== > >= < <=

A comparison operator compares two operands and returns a Boolean value (true or false) as to the validity of the comparison. Operands can be of numeric or string type.

== This is the equal operator and returns a boolean true if both the operands are equal. JavaScript will attempt to convert different data types to the same type in order to make the comparison. Assuming 'a' to be 2 and 'b' to be 4, the following examples will return a value of true:

```
a == 2
a == "2"
2 == '2'
```

!=

This is the not equal operator which returns a Boolean true if both the operands are not equal. Javascript attempts to convert different data types to the same type before making the comparison. The following examples return a Boolean true:

```
a != b
a != 4
a != "2"
```

=== This is the strict equal operator and only returns a Boolean true if both the operands are equal and of the same type. These next examples return true:

```
a === 2
b === 4
```

!==

This is the strict not equal operator and only returns a value of true if both the operands are not equal and/or not of the same type. The following examples return a Boolean true:

```
a !== b
a !== "2"
4 !== '4'
```

>

This is the greater than operator and returns a value of true if the left operand is greater than the right.:

```
a > 1
b > a
```

>= This is the greater than or equal operator, which returns true if the first operand is greater than or equal to the second. The following examples return true:

```
a >= 1
a >= 2
b >= a
```

<

This is the less than operator and returns true if the left operand is less than the right:

```
a < 3
a < b
```

<= This is the less than or equal operator and returns true if the first operand is less than or equal to the second. These next examples all return true:

```
a <= 2
a <= 3
a <= b
```

OPERATORS: && || !

The logical operators are generally used with Boolean values and, when they are, they return a Boolean value. However, both && and || actually return the value of one of their operands, so if the relevent operand is not a Boolean value, the operator may return a non-Boolean value.

&&

This is the logical AND operator, which returns a Boolean true if both the operands are true. Logically it follows that if the first operand is false, then the whole expression is false, and this is how the operator works; It first evaluates the left hand operand, and if this returns false then, without going any further, it returns false for the whole expression. Otherwise it returns the value of the second operand: true or false for a Boolean value, or the actual value itself if non-Boolean. Assuming 'a' to be 3, 'b' to be 5, and 'c' to be 3, the following examples all return true:

```
if((a == c) && (b == 5))
x = "bread" && (c == 3)
```

...while the following return 'cheese':

```
x = (b > c) && "cheese"
x = "bread" && "cheese"
```

||

This is the logical OR operator and it returns a value of true if one or both of the operands is true. It works by first evaluating the left-hand operand and, if this is true, disregarding the right-hand one and returning true for the whole expression. If, however, the left-hand operand is false, then it returns the value of the right-hand operand: true or false if Boolean, or else the value itself. These next examples both return true:

```
if((a == c) || (b == 9))
x = (a > b) || (c == 3)
```

...while this one returns 'cheese':

```
x = (a > b) || "cheese"
```

If, however, the first operand is not a Boolean value, the OR operator returns the value of the first operand whether the second is true or not; in the next example 'bread':

```
x = "bread" || (c == 3)
```

!

This is the logical NOT operator which returns false if its single operand can be converted to true, or if it is a non-Boolean value:

```
x = !(a == c)
x = !"cheese"
```

...and true if its operand can be converted to false:

```
x = !(a > b)
```

OPERATORS: ?: , delete new this typeof void

?:

This is a conditional operator that takes three operands and is used to replace simple **if** statements. The first operand is a condition that evaluates to true or false, the second is an expression of any type to be returned if the condition is true, and third is an expression of any type to be returned if the condition is false. The following code displays one of two messages depending on the value of the object property 'percent_proof':

Code:
```
with(beer)
   document.write("This beer is " + ((percent_proof < 5) ? "mild" : "strong"));
```

,

This is the comma operator which is most often used to include multiple expressions where only one is required, particularly in a **for** loop. It evaluates both its operands and returns the value of the second. The following code uses a two-dimensional array of 3 by 3 elements and initializes two counters (one for each dimension) incrementing them both, which results in a display of the left-to-right diagonal values:

Code:
```
for(var i=0, j=0; i<3; i++, j++)
   document.write(a[i][j] + "<BR>");
```

...while using the comma operator inside the square brackets of the next example causes the code to display all the elements of the array one row at a time:

Code:
```
for(var i=0, j=0; i<3; i++, j++)
   document.write(a[i, j] + "<BR>");
```

delete

The **delete** operator is used to delete an object, an object's property or a specified element in an array, returning true if the operation is possible, and false if not. With the defined object 'fruit' below, the following delete operations are possible:

Code:
```
fruit = new Object;
fruit.name = 'apple';
fruit.color = 'green';
fruit.size = 'large';

delete fruit.size;

with(fruit)
   delete color;

delete fruit;
```

NOTE:

To delete an object's property, you must precede that property's name with the name of the object, unless it's used in a **with** statement.

The **delete** operator can also be used to delete an element of an array. This does not affect the length of the array or any of the other elements but changes the deleted element to undefined. The following example creates an array called 'fruit' and then deletes element #2 (orange):

Code:
```
fruit = new Array ("apple", "pear", "orange", "cherry", "grape");
delete fruit[2];
```

new

The **new** operator can be used to create an instance of a user-defined object type or of one of the built-in object types that has a **constructor** function. To create a user-defined object type you must first define it by writing a function that specifies its name, properties and methods. For example, the following function creates an object for books with properties for title, category and author:

Code:
```
function book(title, category, author)
{
   this.title = title
   this.category = category
   this.author = author
}
```

You can than create an instance of the object as in the following example which assigns the values "The Thing", "horror" and "John Lynch" to the respective properties:

Code:
```
mybook = new book("The Thing", "horror", "John Lynch")
```

The property of an object can even be another object. You could, for example, create an object for authors and use it in the above example of the 'book' object. Again you would first define it by writing a function, and then you could create an instance of it for "John Lynch" as follows:

Code:
```
function author(name, real_name, age)
{
   this.name = name
   this.real_name = real_name
   this.age = age
}
author1 = new author("John Lynch", "Charlie Schwarz", 43)
```

Now, as long as the 'author' object and the relevent instance of it have already been defined, the 'author' property in the 'book' object will refer to it, and you can display, say, the age of the author of 'mybook' as follows:

Code:
```
document.write(mybook.author.age)
```

You can also add a property to any instance of an object as in the next example which adds the property 'publisher' to the 'mybook' object and assigns it the value "Centurion Books":

Code:
```
mybook.publisher = "Centurion Books"
```

If, however, you wanted to add a property to a previously defined object type (and, therefore, all instances of it) you would have to use the **prototype** property. This next example adds the property 'publisher' to the 'book' object type, and then assigns the value 'Centurion Books' to one particular instance of it:

Code:
```
book.prototype.publisher = null
mybook.publisher = "Centurion Books"
```

this

The keyword **this** is used to refer to the current object. In a method, it usually refers to the calling object. In the following example the code first creates a function DescribeAge that takes as its parameter an object, and returns one of two text values depending on the value of that same object's Year property. Then another object called Car is created whose third property Description is the value returned by the DescribeAge function. The keyword **this** is used as the parameter of the DescribeAge function to refer to whichever object is calling it, as seen in the final bit of code which creates a specific instance of the Car object whose Description property will now contain the string "Old-fashioned":

Code:
```
function describeAge(obj)
{
  if(obj.year < 1996)
    return "Old-fashioned"
  else
    return "Good-as-new"
}

function car(make, year, description)
{this.make = make, this.year = year, this.description = describeAge(this)}

myCar = new car("Ford", "1993", describeAge(this))
```

typeof

The **typeof** operator returns the type of an unevaluated operand which can be a number, string, variable, object or keyword. It can be used with or without brackets as in the following examples of a numeric literal and the variable 'age' being 60:

typeof(age) returns number
typeof 33 returns number

The following values are returned for various types of data:

a number returns 'number';
a string returns 'string';
the keyword **true** returns 'boolean';
the keyword **null** returns 'object';
methods, functions and predefined objects return 'function';

a variable returns the type of the data assigned to it;
a property returns the type of its value;

void

The **void** operator evaluates an expression without returning a value. Although the use of brackets after it is optional, it is good style to use them. The following example creates a hyperlink on the word "green" which, when clicked, changes the background color to light green:

Code:
```
Sam turned <a href="javascript:void(document.bgColor='lightgreen')">green</a>.
```

STATEMENT: if...else

if (condition) {statements1} [else {statements2}]

The **if...else** statement executes one set of statements if a specified condition is true, and another if it is false.

The following example tests the result of a function and displays one of two messages depending on whether the result is less than 10 or not:

Code:

```
if(calcaverage(x,y,z) < 10)
   document.write("The average is less than 10.");
else
   document.write("The average is 10 or more.");
```

NOTE:

You shouldn't use simple assignment statements such as **if(a = b)** in a conditional statement.

STATEMENT: For

for ([initial-expression]; [condition]; [increment-expression])
{statements}

The **for** statement creates a loop consisting of three optional expressions enclosed in brackets and separated by semicolons, and a block of statements to be executed. The first expression is used to initialise a counter variable, the second (optional) provides a condition that is evaluated on each pass through the loop, and the third updates or increments the counter variable.

This example simply counts up from zero for as long as the counter is less than 10:

Code:
```
for(i=0; i<10; i++)
   document.write(i + ".<BR>");
```

STATEMENT: with

with

The **with** statement establishes a default object for a set of statements. If there are any unqualified names in a set of statements, JavaScript first checks the default object to see if they exist there as properties of that object; otherwise a local or global variable is used.

In the following example the default object is 'beer' and the code displays one of two messages depending on the value of the property 'percent_proof':

Code:
```
with(beer)
{
   if(percent_proof < 5)
      document.write("Call this a strong beer?!");
   else
      document.write("This is what I call a beer!");
}
```

PROPERTY: Object::constructor

Object.constructor

A **constructor** property is inherited by all objects from their prototype. It is this fact that allows you to create a new instance of an object using the **new** operator. If you display the **constructor** property of any object, you see the construction of the function that created it.

For example, assuming the existence of an object called 'Cat', the following code would display that function:

Code:
```
document.write(Cat.constructor)
```

Output:
```
function Cat(breed, name, age) { this.breed = breed this.name = name
this.age = age }
```

The **constructor** property can also be used to compare two objects (including those, such as documents and forms, that cannot be constructed). This next example compares the 'Sheeba' object with the 'Cat' object to see if it is an instance of it:

Code:
```
if(Sheeba.constructor == Cat)
   document.write("This is an instance of 'Cat'.")
```

OPERATORS: +

+

The **+** plus string operator is used to concatenate (join together) two or more strings. This example joins together the three strings 'bread', 'and' and 'cheese' to produce 'bread and cheese':

```
document.write("bread " + "and " + "cheese");
```

...while the next one would, assuming 'x' to contain the string 'honey', return 'milk and honey':

```
document.write("milk and " + x);
```

The shorthand assignment operator can also be used to concatenate strings. If the variable 'x' has the value 'milk and ', then the following code will add the string 'honey' to the value in variable 'x' producing the new string 'milk and honey':

Code:
```
var x = 'milk and ';
x += "honey";
document.write(x);
```

Output:
```
milk and honey
```

PROPERTY: RegExp::$1, ..., $9

These are properties containing parenthesized substrings (if any) from a regular expression. The number of parenthesized substrings is unlimited, but these properties only hold the last nine. All parenthesized substring, however, can be accessed throught the returned array's indexes. When used as the second argument of the **String.replace** method, these properties do not require **RegExp.** before them. In the following example, the day and month parts of a date written in British format are rearranged to American style:

Code:

```
myRegExp = /(\d{2})\W(\d{2})\W(\d{4})/
dateString = "25/12/1997"
newString = dateString.replace(myRegExp, "$2/$1/$3")
document.write(newString)
```

Output:

```
12/25/1997
```

NOTE:

Because input is static, it is always used as **RegExp.input**

OBJECT: RegExp

new RegExp("pattern"[, "flags"])

The **RegExp** object contains the pattern of a regular expression, and is used to match strings using its methods and properties. The predefined **RegExp** object has static properties set whenever a regular expression is used, as well as user-defined ones for individual objects. A **RegExp** object can be created in two ways: either by using the **constructor** function, or with a literal text format. In both cases you need to specify the text pattern of the regular expression, and optionally one of the three possible flags: 'g' for a global match, 'i' to ignore case, or 'gi' for a case-insensitive global match.

The following code creates a regular expression using the **constructor** function, that matches an initial letter 'a' irrespective of case:

Code:
```
myRegExp = new RegExp("^a", "i")
```

Note that when using the **constructor** function, you must use quotation marks to indicate strings. Also, the normal string escape rules apply and you must use a back slash before special characters as in the following code which creates a regular expression of a tab character:

Code:
```
myRegExp = new RegExp("\\t")
```

Using the literal text format you do not need to use quotation marks. To create a regular expression consisting of a case-insensitve initial 'a' you could use the following code:

Code:
```
myRegExp = /^a/i
```

...and a regular expression consisting of a tab character could be created as follows:

Code:
```
myRegExp = /\t/
```

The literal notation of a regular expression provides compilation of it when the expression is evaluated, and this is used when you know that a regular expression is going to remain constant. When you know that an expression is going to change, or you don't know what that expression is because it will be input by a user, you need to use the **constructor** function which is compiled at run time. You can, however, compile a regular expression at any time using the **compile** method. Each window has its own predefined **RegExp** object thus ensuring that different threads of JavaScript execution don't overwrite values of the **RegExp** object.

For a complete list and description of the special characters that can be used in a regular expression, see the **special characters** page.

PROPERTIES

$1, ..., $9 Property
These are properties containing parenthized substrings (if any) from a regular expression.

$ Property
See the **input** property.

$* Property
See the **multiline** property.

$& Property
See the **lastMatch** property.

$+ Property
See the **lastParen** property.

$` Property
See the **leftContext** property.

$' Property
See the **rightContext** property.

constructor Property
This property specifies the function that creates an object's prototype. See the **Object.constructor** property.

Syntax: **RegExp.constructor**

global Property
This property reflects whether the 'g' flag was used to match a regular expression globally in a string, or just the first occurrence of it. Its value is **true** if the 'g' flag was used and **false** if not. Note that this property is read-only but that calling the **compile** method does alter it.

Syntax: **object.global**

ignoreCase Property
This property reflects whether the 'i' flag was used for a case-insensitive match of a regular expression in a string, returning **true** if it was and **false** if not. Note that this property is read-only but that calling the **compile** method does alter it.

Syntax: **object.ignoreCase**

input Property
This property is a string against which a regular expression is matched.

Syntax: **RegExp.input**

lastIndex Property
This property is an integer that specifies the index at which to start the next match, but is only set if the regular expression uses the 'g' flag to specify a global search.

Syntax: **object.lastIndex**

lastMatch Property

This property is the last matched characters. As this property is static, you always use **RegExp.lastMatch**.

Syntax: **RegExp.lastMatch**

lastParen Property

This property contains the last matched parenthesized substring (if any), and as a static property is always refered to using **RegExp.lastParen**.

Syntax: **RegExp.lastParen**

leftContext Property

This property is the substring upto the character most recently matched; i.e. everything that comes before it, and as a static property, is always used as **RegExp.leftContext**.

Syntax: **RegExp.leftContext**

multiline Property

This property reflects whether a search is to be carried out over multiple lines, returning **true** if it is, and **false** if not. Being a static property, you always use **RegExp.multiline**. When an event handler is called for a TEXTAREA form element, the browser sets the **multiline** property to **true**. Once the event handler has finished executing, it is reset to **false**, even if it was set at **true** before the event handler was called.

Syntax: **RegExp.multiline**

prototype Property

This property represents the prototype for this class, and allows you to add your own properties and methods to all instances of it. See the **Function.prototype** property.

Syntax: **RegExp.prototype**

rightContext Property

This property is the substring after the character most recently matched; i.e. everything that follows it, and as a static property, is always used as **RegExp.rightContext**.

Syntax: **RegExp.rightContext**

source Property

This is a read-only property containing the source of the regular expression: i.e. everything except the forward slashes and any flags. The **source** property cannot be changed directly, however calling the **compile** method does alter it. For example, with the regular expression rexp = /[^aeiou\s]{2}/g the value of the **source** property would be [^aeiou\s]{2}.

Syntax: **object.source**

METHODS

compile Method
This method compiles a regular expression object during execution of a script.

Syntax: **object.compile(pattern[, flags])**

exec Method
This method executes a search for a match in a specified string, returning a result array.

Syntax: **object.exec([str])**

object([str])

test Method
This method tests for a match of a regular expression in a string, returning **true** if successful, and **false** if not.

Syntax: **object.test([str])**

toSource Method
This method returns the source code of a **RegExp** object and is usually called internally by JavaScript. It also overrides the **Object.toSource** method.

Syntax: **object.toSource()**

toString Method
This method returns a string representing the **RegExp** object, and this overrides the **Object.toString method**.

Syntax: **object.toString()**

valueOf Method
This method returns a primitive value for the **RegExp** object as a string data type, and is equivalent to the **RegExp.toString** method. It is usually called internally by JavaScript and overrides the **Object.valueOf** method.

Syntax: **object.valueOf()**

NOTE:

The **regExp** object also inherits the **watch** and **unwatch** methods from the **Object** object.

Special Characters

The following is a complete list of all the special characters that can be used in a regular expression. Note that the flags 'g', 'i' and 'gi' can be used after the final slash to specify a global, case-insensitive or global, case-insensitive search respectively. See the **RegExp** object.

\

The backslash is used in two ways. Firstly it is used before any letter of the alphabet when not used literally, but to indicate a special character. For example, a regular expression consisting of the letter 't' would be created using /t/, whereas for a tab character you would use /\t/. Secondly it is used before a special character which you want to use literally e.g. the character $, which is used to match a character at the end of a line or of input, would become \$ when used literally.

^

The caret is used for a match at the beginning of a line or of input. For example, with the string "Association of Carpenters", the regular expression /^A/ would match the initial capital letter 'A' of Association' but not the initial 'A' of 'Association' in the string "Teachers Association".

$

The dollar sign is used for a match at the end of a line or of input. So, with the string "his cats" the regular expression /s$/ would match the final letter 's' in 'cats' but not in 'his'.

*****

The asterisk is used to match 0 or more occurrences of the preceding character. So, for example, the regular expression /ators*/ would match the 'ator' of "alligator" and the 'ators' of "navigators". However, the regular expression /a*/g, where the asterisk follows a single letter, would match 1 or more occurrences of the letter 'a' throughout the string

+

The plus sign matches one or more occurrences of the preceding character the first time it appears in a string. As such, it is equivalent to {1,}. For example, the regular expression /e+/ would match the 'e' in 'sped' and the 'e' in 'speed'.

?

The question mark is used to match the preceding character 0 or 1 times. For example, the regular expression /e?re?/ would match the string "theater" and also the British spelling "theatre".

.

The decimal point matches any single character except the new line character. So, for instance, with the string "The cat eats moths" the regular expression /.t/gi would match the letters 'at' in 'cat', 'at' in 'eats' and 'ot' in 'moths', but not the initial 'T' of 'The'.

(x)

Putting a regular expression inside parens causes it to be matched and remembered. Each bracketed expression can then be referenced using the index of the resulting array, or by using the **$1, ..., $9** property of the **RegExp** object.

x|y

This expression matches either x or y, so, for example, the regular expression /hot|cold/ will match the 'hot' of 'hot potato' and the 'cold' of 'cold potato'.

{n}

This expression matches 'n' occurrences of the preceding character, where 'n' is an integer. So with the string "the missing snake hisssed" the regular expression /s{2}/g would match both the 'ss' in 'missing' and the first two of the three 's's in 'hisssed'.

{n,}

This expression matches at least n occurrences of the preceding character, where 'n' is a positive integer. With the string "the missing snake hisssed", the regular expression /s{2,}/g/ would match the 'ss' of 'missing' and the 'sss' of 'hisssed'.

{n,m}

This expression matches at least 'n' and at most 'm' occurrences of the preceding character, where 'n' and 'm' are positive integers. So, with the string "the missing snake hissssssed", the regular expression /s{2,4}/g will match the 'ss' of 'missing' and the first four 's's of 'hissssssed'.

[xyz]

This expression matches any one of the set of characters enclosed within the square brackets. A series of characters can be separated by a hyphen: e.g. you can use [abcde] or [a-e]. With the string "the black cat", the regular expression /[abc]/g would match the 'b', 'a' and 'c' in 'black' and the 'c' and 'a' in 'cat'.

[^xyz]

This expression matches any character other than those following the caret. A series of characters can be separated by a hyphen: e.g. you can use [a-e] instead of [abcde]. With the string "black", the regular expression /[^bla]/ would match the letter 'c'.

[\b]

This expression matches a backspace character.

\b

This expression matches any word boundary such as a space. So, for instance, with the string "the black cat", the regular expression /\bc/ would match the 'c' in 'cat', not the one in 'black'.

\B

This expression is used to match a non-word boundary. For example, with the string "the black cat", the regular expression /\Bc/g would match the 'c' in 'black' but not the one in 'cat'.

\cX

This expression matches a control character: i.e. a combination of the contol key <CTRL> and any other key represented by the 'X'.

\d

This expression matches any digit character, and is equivalent to [0-9]. For example, with the string "the 4th of July", the regular expression /\d/ would match the character '4'. (The expression /[0-9]/ would work just the same.)

\D

This expression matches any non-digit character, and is equivalent to [^0-9]. So, with the string "45X", the regular expression /\D/ would match the letter 'X'. (The expression [^0-9] would work just as well.

\f

This expression matches a formfeed.

\n

This expression matches a linefeed.

\r

This expression matches a carriage return.

\s

This expression matches any white space character including tab, line feed and form feed. It is equivalent to [\t\v\f\n\r]. So, for example, with the string "Christmas Day", the expression /\s/ would match the space between the two words.

\S

This expression is the opposite to \s and matches any non-white space character. it is equivalent to [^ \t\v\f\n\r]. So, with the string "Christmas Day", the expression /\S/ would match the 'C' of 'Christmas'.

\t

This expression matches a tab character.

\v

This expression matches a vertical tab character.

\w

This expression matches any alpha-numeric character including the underscore, and is therefore equivalent to [a-zA-Z0-9_]. For example, with the string "O = 2a", the regular expression /\w/ would match the character '2'.

\W

This expression is the opposite of \w and matches any character other than an alpha-numeric character or the underscore. It is therefore equivalent to [^a-zA-Z0-9_]. With the string "O = 2a", the expression /\W/ matches the character 'O'.

\n

Where n is a positive integer, this expression refers to a previous parenthesized substring within a regular expression. (See the **(x)** expression.) With the string "John, Francis, Bob, Sherri", the regular expression /[E-H]\w*(,)(\s)\w*\1\2/ would match the substring "Francis, Bob, ". This saves you having to type a complicated substring repeated within a regular expression.

Note that if n represents a number higher than the number of previous parenthesized substrings, then it is treated as an octal return. See below.

\ooctal

Where o is an octal escape value, this expression allows you to embed ASCII codes into regular expressions.

\xhex

Where x is an hexadecimal escape value, this expression allows you to embed ASCII codes into regular expressions.

PROPERTY: RegExp::input

This property is a string against which a regular expression is matched. Input is static and is therefore always used as **RegExp.input**. If this property has a value, that value is used by default as the argument of the **exec** and **test** methods, unless another argument is provided. When no argument is provided with these methods, the script or browser can preset the **input** method as follows:

Where a TEXT form element calls an event handler, it is set to the value of the contained text.

Where a TEXTAREA form element calls an event handler, it is set to the value of the contained text and the **multiline** property is set to true so that the match can be executed throughout the text.

Where a SELECT form element calls an event handler, it is set to the value of the selected text.

Use the **href** property to change a link. Where a **Link** object calls an event handler, it is set to the value of the text between the <A> tags.

The value of **input** clears after the event handler completes.

PROPERTY: Link::href

object.href

href is a property of both the **Link** and the **Location** objects and is a string specifying the entire URL, of which all other properties are substrings. If you wish to change a location, you can safely do so by setting the **href** property. Assuming a link (stored in the first element of the **links** property) to a URL ...

http://home.newco.com/products/enquiries.htm#local?email=name@otherco.com

...the following code would display the **href** property of it:

Code:
```
document.write(document.links[0].href)
```

Output:
```
http://home.newco.com/products/enquiries.htm#local?email=name@otherco.com
```

OBJECT: Link

The **Link** object is a piece of text, an image or an area of an image that loads a hypertext link reference into the target window when selected. **Area** objects are also a type of **Link** object. A link can be created either by using the HTML 'A' or 'AREA' tags, or by calling the **String.link** method. Each 'A' or 'AREA' tag that has an HREF attribute is placed by the JavaScript engine in an array in the **document.links** property. A **Link** object can then be accessed by indexing this array. The following code demonstrates the creation of a link to an anchor in the 'Authors.htm' page using the **String.link** method:

Code:
```
document.write("AUTHORS".link("Authors.htm#author"))
```

The exact same link can also be created using HTML as follows:

Code:
```
<a href="Authors.htm#author">AUTHORS</a>
```

A **Link** object is a **Location** object and shares the same properties. When you click a **Link** object, the destination document's **referrer** property then contains the URL of the source page. A link can also be used to execute JavaScript code rather than to reference a hyperlink. The following code, for example, simply creates a function to display the message "Hello World!" which is then called if the user clicks on the 'GREETINGS' link:

Code:
```
<script language="javascript">
function write_hello()
{
  document.write("Hello World!")
}
</script>
<a href="javascript:write_hello()">GREETINGS</a>
```

A full URL takes the following form:

<protocol>//<host>[:<port>]/<pathname>[<hash>][<search>]

PROPERTIES

hash Property

The **hash** property is a string beginning with a hash (#), that specifies an anchor name in an HTTP URL.

Syntax: **object.hash**

host Property

The **host** property is a string comprising of the **hostname** and **host** strings.

Syntax: **object.host**

hostname Property

The **hostname** property specifies the server name, subdomain and domain name (or IP address) of a URL.

Syntax: **object.hostname**

href Property
The **href** property is a string specifying the entire URL, and of which all other **Link** properties are substrings.

Syntax: **object.href**

pathname Property
The **pathname** property is a string portion of a URL specifying how a particular resource can be accessed.

Syntax: **object.pathname**

port Property
The **port** property is a string specifying the communications port that the server uses.

Syntax: **object.port**

protocol Property
The **protocol** property is the string at the beginning of a URL, up to and including the first colon (:), which specifies the method of access to the URL.

Syntax: **object.protocol**

search Property
The **search** property is a string beginning with a question mark that specifies any query information in an HTTP URL.

Syntax: **object.search**

target Property
The **target** property is a string specifying the window that displays the contents of a clicked hyperlink.

Syntax: **object.target**

text Property
The **text** property is a string containing the text of a corresponding 'A' tag.

Syntax: **object.text**

METHODS

handleEvent Method
The **HandleEvent** method invokes the handler for the specified event.

Syntax: **object.handleEvent(event)**

EVENT HANDLERS

The **Link** object has all of the following event handlers, but the **Area** object can only use the **onDblClick**, **onMouseOut** and **onMouseOver** event handlers.

onClick EventHandler

The **onClick** event handler executes javaScript code whenever the user clicks (i.e. when the mouse button is pressed and released) on a form object.

Syntax: **onClick = "myJavaScriptCode"**

onDblClick EventHandler

The **onDblClick** event handler executes JavaScript code whenever the user double clicks on an object in a form.

Syntax: **onDblClick = "myJavaScriptCode"**

onKeyDown EventHandler

The **onKeyDown** event handler is used to execute certain JavaScript code whenever the user depresses a key.

Syntax: **onKeyDown = "myJavaScriptCode"**

onKeyPress EventHandler

The onKeyPress event handler executes JavaScript code whenever the user presses or holds down a key

Syntax: **onKeyPress = "myJavaScriptCode"**

onKeyUp EventHandler

The **onKeyUp** event handler executes JavaScript code whenever the user releases a depressed key.

Syntax: **onKeyUp = "myJavaScriptCode"**

onMouseDown EventHandler

The **onMouseDown** event handler executes JavaScript code whenever the user depresses the mouse button over an area or link.

Syntax: **onMouseDown = "myJavaScriptCode"**

onMouseOut EventHandler

The **onMouseOut** event handler executes JavaScript code whenever the mouse pointer leaves an area or a link within that area or link.

Syntax: **onMouseOut = "myJavaScriptCode"**

onMouseUp EventHandler

The **onMouseUp** event handler executes JavaScript code whenever the user releases a depressed mouse button over an area or link.

Syntax: **onMouseUp = "myJavaScriptCode"**

onMouseOver EventHandler

The **onMouseOver** event handler cause JavaScript code to be executed whenever the mouse pointer leaves an area or a link within an area or link.

Syntax: **onMouseOver "myJavaScriptCode"**

PROPERTY: Link::hash

object.hash

hash is a property of both the **Link** and the **Location** objects and is a string beginning with a hash (#). It specifies an anchor name in an HTTP URL. Assuming a link (stored in the first element of the **links** property) to a URL ...

http://home.newco.com/products/enquiries.htm#local?email=name@otherco.com

...the following code would display the **hash** property of it:

Code:
```
document.write(document.links[0].hash)
```

Output:
```
#local?email=name@otherco.com
```

NOTE:

Although the **hash** property can be set at any time, it is safer to set the **href** property to change a link.

OBJECT: Location

The **Location** object is part of a **Window** object and is accessed through the **window.location** property. It contains the complete URL of a given **Window** object, or, if none is specified, of the current **Window** object. All of its properties are strings representing different portions of the URL, which generally takes the following form:

<protocol>//<host>[:<port>]/<pathname>[<hash>][<search>]

You can create a **Location** object by simply assigning a URL to the **location** property of an object:

Code:
```
window.location = "file:///C:/Projects"
```

PROPERTIES

hash Property
The **hash** property is a string beginning with a hash (#), that specifies an anchor name in an HTTP URL.

Syntax: **location.hash**

host Property
The **host** property is a string comprising of the **hostname** and **port** strings.

Syntax: **location.host**

hostname Property
The **hostname** property specifies the server name, subdomain and domain name (or IP address) of a URL.

Syntax: **location.hostname**

href Property
The **href** property is a string specifying the entire URL, and of which all other **Link** properties are substrings.

Syntax: **location.href**

pathname Property
The **pathname** property is a string portion of a URL specifying how a particular resource can be accessed.

Syntax: **location.pathname**

port Property
The **port** property is a string specifying the communications port that the server uses.

Syntax: **location.port**

protocol Property

The **protocol** property is the string at the beginning of a URL, up to and including the first colon (:), which specifies the method of access to the URL.

Syntax: **location.protocol**

search Property

The **search** property is a string beginning with a question mark that specifies any query information in an HTTP URL.

Syntax: **location.search**

METHODS

reload Method

The **reload** method forces a reload of the window's current document, i.e. the one contained in the **Location.href** property.

Syntax: **location.reload([forceGet])**

replace Method

The **replace** method replaces the current **History** entry with the specified URL. After calling the **replace** method, you cannot navigate back to the previous URL using the browser's Back button.

Syntax: **location.replace(URL)**

PROPERTY: Location::hash

location.hash

hash is a property of both the **Link** and the **Location** objects and is a string beginning with a hash (#). It specifies an anchor name in an HTTP URL. For example, assuming the following URL to be your current window...

http://home.newco.com/products/enquiries.htm#local?email=name@otherco.com

...you could find out the **hash** property as follows:

Code:
```
document.write(location.hash)
```

Output:
```
#local?email=name@otherco.com
```

NOTE:

Although the **hash** property can be set at any time, it is safer to set the **href** property to change a location

PROPERTY: Location::href

location.href

href is a property of both the **Link** and the **Location** objects and is a string specifying the entire URL, of which all other properties are substrings. If you wish to change a location, you can safely do so by setting the **href** property. Assuming your current window to be the URL...

http://home.newco.com/products/enquiries.htm#local?email=name@otherco.com

...you could find out the **href** property as follows:

Code:
```
document.write(location.href)
```

Output:
```
http://home.newco.com/products/enquiries.htm#local?email=name@otherco.com
```

PROPERTY: Location::host

location.host

host is a property of both the **Link** and the **Location** objects and is a string comprising of the **hostname** and **port** strings. If the **port** property is the default of 80, then the **hostname** property is the same as the **host** property. Assuming your current window to be the URL...

http://home.newco.com/products/enquiries.htm#local?email=name@otherco.com

...you could find out the **host** property as follows:

Code:
```
document.write(location.host)
```

Output:
```
home.newco.com:80
```

NOTE:

Although the **host** property can be set at any time, it is safer to set the **href** property to change a location.

PROPERTY: Location::hostname

location.hostname

hostname is a property of both the **Link** and the **Location** objects and specifies the server name, subdomain and domain name (or IP address) of a URL. Assuming your current window to be the URL...

http://home.newco.com/products/enquiries.htm#local?email=name@otherco.com

...you could find out the **hostname** property as follows:

Code:
```
document.write(location.hostname)
```

Output:
```
home.newco.com
```

NOTE:

Although the **hostname** property can be set at any time, it is safer to set the **href** property to change a location.

PROPERTY: Location::pathname

location.pathname

pathname is a property of both the **Link** and the **Location** objects and is a string portion of a URL specifying how a particular resource can be accessed. Assuming your current window to be the URL...

http://home.newco.com/products/enquiries.htm#local?email=name@otherco.com

...you could find out the **pathname** property as follows:

Code:
```
document.write(location.pathname)
```

Output:
```
products/enquiries.htm
```

NOTE:

Although the **pathname** property can be set at any time, it is safer to set the **href** property to change a location.

PROPERTY: Location::port

location.port

port is a property of both the **Link** and the **Location** objects and is a string specifying the communications port that the server uses. If this is the default of 80, then it is not specified, with the result that the **hostname** property will be the same as the **host** property. Assuming your current window to be the URL...

http://home.newco.com/products/enquiries.htm#local?email=name@otherco.com

...you could find out the **port** property as follows:

Code:
```
document.write(location.port)
```

Output:
```
80
```

NOTE:

Although the **port** property can be set at any time, it is safer to set the **href** property to change a location.

PROPERTY: Location::protocol

location.protocol

protocol is a property of both the **Link** and the **Location** objects and is the string at the beginning of a URL, up to and including the first colon (:), which specifies the method of access to the URL. Assuming your current window to be the URL...

http://home.newco.com/products/enquiries.htm#local?email=name@otherco.com

...you could find out the **protocol** property as follows:

Code:

```
document.write(location.protocol)
```

Output:

```
http:
```

NOTE:

Although the **protocol** property can be set at any time, it is safer to set the **href** property to change a location.

PROPERTY: Location::search

location.search

search is a property of both the **Link** and the **Location** objects and is a string beginning with a question mark that specifies any query information in an HTTP URL. It contains one or more pairs of variables and values, separated by ampersands. Assuming your current window to be the URL...

http://home.newco.com/products/enquiries.htm#local?email=name@otherco.com

...you could find out the **search** property as follows:

Code:

```
document.write(location.search)
```

Output:

```
?email=name@otherco.com
```

NOTE:

Although the **search** property can be set at any time, it is safer to set the **href** property to change a location.

METHOD: Location::reload

Location.reload([forceGet])

The **reload** method forces a reload of the window's current document, i.e. the one contained in the **Location.href** property. It behaves in exactly the same way as the browser's reload button which, by default reloads from the cache. If, however, the user has specified that the server be checked every time for an updated version, then the **reload** method will do the same. You can force an unconditional HTTP GET of the document from the server by supplying 'true' for the parameter, but this should only be done if you believe that disk and memory caches are off or broken, or you believe the server has an updated version.

In the following example the event handler calls the **reload** method in response to the clicking of a button:

Code:
```
onClick="window.location.reload()"
```

PROPERTY: Link::host

object.host

host is a property of both the **Link** and the **Location** objects and is a string comprising of the **hostname** and **port** strings. If the **port** property is the default of 80, then the **hostname** property is the same as the **host** property. Assuming a link (stored in the first element of the **links** property) to a URL ...

http://home.newco.com/products/enquiries.htm#local?email=name@otherco.com

...the following code would display the **host** property of it:

Code:
```
document.write(document.links[0].host)
```

Output:
```
home.newco.com:80
```

NOTE:

Although the **host** property can be set at any time, it is safer to set the **href** property to change a link.

PROPERTY: Link::hostname

object.hostname

hostname is a property of both the **Link** and the **Location** objects and specifies the server name, subdomain and domain name (or IP address) of a URL. Assuming a link (stored in the first element of the **links** property) to a URL ...

http://home.newco.com/products/enquiries.htm#local?email=name@otherco.com

...the following code would display the **hostname** property of it:

Code:

```
document.write(document.links[0].hostname)
```

Output:

```
home.newco.com
```

NOTE:

Although the **hostname** property can be set at any time, it is safer to set the **href** property to change a link.

PROPERTY: Link::pathname

object.pathname

Pathname is a property of both the **Link** and the **Location** objects and is a string portion of a URL specifying how a particular resource can be accessed. Assuming a link (stored in the first element of the **links** property) to a URL ...

http://home.newco.com/products/enquiries.htm#local?email=name@otherco.com

...the following code would display the **pathname** property of it:

Code:

```
document.write(document.links[0].pathname)
```

Output:

```
products/enquiries.htm
```

NOTE:

Although the **pathname** property can be set at any time, it is safer to set the **href** property to change a link.

PROPERTY: Link::port

object.port

Port is a property of both the **Link** and the **Location** objects and is a string specifying the communications port that the server uses. If this is the default of 80, then it is not specified, with the result that the **hostname** property will be the same as the **host** property. Assuming a link (stored in the first element of the **links** property) to a URL ...

http://home.newco.com/products/enquiries.htm#local?email=name@otherco.com

...the following code would display the **port** property of it:

Code:
```
document.write(document.links[0].port)
```

Output:
```
80
```

NOTE:

Although the **port** property can be set at any time, it is safer to set the **href** property to change a link.

PROPERTY: Link::protocol

object.protocol

Protocol is a property of both the **Link** and the **Location** objects and is the string at the beginning of a URL, up to and including the first colon (:), which specifies the method of access to the URL. Assuming a link (stored in the first element of the Links property) to a URL ...

http://home.newco.com/products/enquiries.htm#local?email=name@otherco.com

...the following code would display the **protocol** property of it:

Code:
```
document.write(document.links[0].protocol)
```

Output:
```
http:
```

NOTE:

Although the **protocol** property can be set at any time, it is safer to set the **href** property to change a link

PROPERTY: Link::search

object.search

Search is a property of both the **Link** and the **Location** objects and is a string beginning with a question mark that specifies any query information in an HTTP URL. It contains one or more pairs of variables and values, separated by ampersands. Assuming a link (stored in the first element of the **links** property) to a URL ...

http://home.newco.com/products/enquiries.htm#local?email=name@otherco.com

...the following code would display the **search** property of it:

Code:
```
document.write(document.links[0].search)
```

Output:
```
?email=name@otherco.com
```

NOTE:

Although the **search** property can be set at any time, it is safer to set the **href** property to change a location.

PROPERTY: Link::target

object.target

The **target** property is a string specifying the name of the window or frame that displays the contents of a clicked hyperlink. If you provide a name that does not exist, a new window will be opened. By default this is the target attribute of the <A> or <AREA> tags, but you can override it by setting the **target** property. You can do this at any time, but you cannot assign the value of a JavaScript expression or variable.

EVENT HANDLER: onDblClick

onDblClick = myJavaScriptCode

Event handler for **Document**, **Link**.

The **onDblClick** event handler executes the specified JavaScript code or function on the occurance of a double click event.

The **onDblClick** event handler uses the following **Event** object properties.

type - this property indicates the type of event.
target - this property indicates the object to which the event was originally sent.
layerX - the cursor location when the click event occurs.
layerY - the cursor location when the click event occurs.
pageX - the cursor location when the click event occurs.
pageY - the cursor location when the click event occurs.
screenX - the cursor location when the click event occurs.
screenY - the cursor location when the click event occurs.
which - 1 represents a left mouse double-click and 3 a right double-click.
modifiers - lists the modifier keys (shift, alt, ctrl, etc.) held down when the click event occurs.

The following example shows the use of the **onDblClick** event handler with a **document** object. As you can see, this is defined within the document's HTML <BODY> tag.

Code:

```
<body onDblClick = "document.write('Hello World!')">
```

NOTE:

Macintosh platforms don't support mouse double-clicks and therefore this event handler is not implemented.

OBJECT: Document

The **Document** object provides access to the elements in an HTML page from within your script. This includes the properties of every form, link and anchor (and, where applicable, any sub-elements), as well as global **Document** properties such as background and foreground colors.

PROPERTIES

alinkColor Property
This property defines the color of an active link. The **"colorinfo"** argument is a string that can contain either the hexadecimal definition of the color or it's literal description.

Syntax: **document.alinkColor = "colorinfo"**

anchors Property
This property is an array containing references to all the named **Anchor** objects in the current document.

Syntax: **document.anchors["anchorID"]**

applets Property
This property is an array containing references to all the **Applet** objects in the current document.

Syntax: **document.applets["appletID"]**

bgColor Property
This property defines a document's background color. The **"colorinfo"** argument is a string that can contain either the hexadecimal definition of the color or it's literal description.

Syntax: **document.bgColor = "colorinfo"**

cookie Property
This property is a string that returns a report detailing all visible and un-expired cookies that are associated with the specified document.

Syntax: **document.cookie [= "expression(s)"]**

domain Property
This property sets or returns the domain name of the server from which the document originated.

Syntax: **document.domain = "domaininfo"**

embeds Property
This property is an array containing references to all the embedded objects in the current document.

Syntax: **document.embeds["embed_objID"]**

fgColor Property
This property defines a document's foreground (text) color. The **"colorinfo"** argument is a string that can contain either the hexadecimal definition of the color or it's literal description.

Syntax: **document.fgColor = "colorinfo"**

formName Property
Every form in a document has a separate **document** object property, the name of which is taken from the value asigned to the form with the <FORM NAME = "formID"> tag. Any form in the document can then be referred to with the syntax below.

Syntax: **document."formname"**

forms Property
This property is an array containing references to all the **Form** objects in the current document.

Syntax: **document.forms["formID"]**

images Property
This property is an array containing references to all the **Image** objects in the current document.

Syntax: **document.images["imageID"]**

lastModified Property
This property returns the date that the document was last modified.

Syntax: **document.lastModified**

layers Property
This property is an array containing references to all the **Layer** objects in the current document.

Syntax: **document.layers["layerID"]**

linkColor Property
This property defines the color of any hyperlinks in the document. The **"colorinfo"** argument is a string that can contain either the hexadecimal definition of the color or it's literal description.

Syntax: **document.linkColor = "colorinfo"**

links Property
This property is an array containing references to all the **Area** and **Link** objects in the current document.

Syntax: **document.links["linkID"]**

plugins Property
This property is an array containing references to all the **Plugin** objects in the current document.

Syntax: **document.plugins["pluginID"]**

referrer Property
If a destination document is reached by a user clicking on a **Link** object in another document (the referrer), this property returns the referring document's URL.

Syntax: **document.referrer**

title Property

This property returns the document's name as defined between the <TITLE></TITLE> tags.

Syntax: **document.title**

URL Property
This property is used to retrieve the document's full URL.

Syntax: **document.URL**

vlinkColor Property
This property defines the color of any visited links in the document. The **"colorinfo"** argument is a string that can contain either the hexadecimal definition of the color or it's literal description.

Syntax: **document.vlinkColor = "colorinfo"**

METHODS

captureEvents Method
This method instructs the document to capture and handle all events of a particular type. See the **event** object for a list of event types.

Syntax: **document.captureEvents(eventType)**

close Method
This method closes an output stream previously opened with the **document.open** method and forces data collected from any instances of the **document.write** or **document.writeln** methods to be displayed.

Syntax: **document.close( )**

getSelection Method
This method can be used to return the contents of selected text in the current document.

Syntax: **document.getSelection( )**

handleEvent Method
This method calls the handler for the specified event.

Syntax: **document.handleEvent(event)**

open Method
This method is used to open a stream to collect the output from any **write** or **writeln** methods.

Syntax: **document.open([mimeType[, replace]])**

releaseEvents Method
This method is used to set the document to release any events of the type **eventType** and passes them along to objects further down the event heirarchy.

Syntax: **document.releaseEvents(eventType)**

routeEvent Method
This method is used to send the event specified along the normal event hierarchy.

Syntax: **document.routeEvent(event)**

write Method
This method is used to write HTML expressions to the specified document.

Syntax: **document.write("expression(s)")**

writeln Method
This method is identical to the **write** method detailed above, with the addition of writing a new line character after any specified expressions.

Syntax: **document.writeln("expression(s)")**

EVENT HANDLERS

onClick Event handler
This event handler executes some specified JavaScript code on the occurrence of a **Click** event (when an element is clicked).

Syntax: **document.onClick="myJavaScriptCode"**

onDblClick Event handler
This event handler executes some specified JavaScript code on the occurrence of a **DblClick** event (when an element is double-clicked).

Syntax: **document.onDblClick="myJavaScriptCode"**

onKeyDown Event handler
This event handler executes some specified JavaScript code on the occurrence of a **KeyDown** event (when a key is depressed).

Syntax: **document.onKeyDown="myJavaScriptCode"**

onKeyPress Event handler
This event handler executes some specified JavaScript code on the occurrence of a **KeyPress** event (when a key is depressed and held down).

Syntax: **document.onKeyPress="myJavaScriptCode"**

onKeyUp Event handler
This event handler executes some specified JavaScript code on the occurrence of a **KeyUp** event (when a key is released).

Syntax: **document.onKeyUp="myJavaScriptCode"**

onMouseDown Event handler
This event handler executes some specified JavaScript code on the occurrence of a **MouseDown** event (when a mouse button is depressed).

Syntax: **document.onMouseDown="myJavaScriptCode"**

onMouseUp Event handler
This event handler executes some specified JavaScript code on the occurrence of a **MouseUp** event (when a mouse button is released).

Syntax: **document.onMouseUp="myJavaScriptCode"**

OBJECT: Button

---

document.alinkColor = "colorinfo"

This property defines the color of an active link (defined as after mouse button down, but before mouse button up). The **"colorinfo"** argument is a string that can contain either the hexadecimal definition of the color or its literal description. If you use the hex definition of a color it must be in the format rrggbb - for example, the hex value for the named color 'forest green' is '228B22'.

Both lines in the follwing code do exactly the same thing, the first using the hex value of a color and the second using its name.

Code:
```
document.alinkColor = "228B22"
document.alinkColor = "forestgreen"
```

PROPERTY: Document::anchors

document.anchors["anchorID"]

This property is an array containing references to all the named **Anchor** objects in the current document. These references are stored in the array in the order in which they are defined in the source code. The **"anchorID"** argument is used to access items in the array and this can either be a string containing the anchor name as defined within the <A></A> tags in the HTML source, or an integer (with '0' being the first item in the array).

Both examples below return the same results; the first uses the defined names of the anchors and the second uses their reference number within the array.

Code:
```
document.anchors["anchorname1"]
document.anchors["anchorname2"]
document.anchors["anchorname3"]

document.anchors[0]
document.anchors[1]
document.anchors[2]
```

PROPERTY: Document::applets

document.applets["appletID"]

This property is an array containing references to all the named **Applet** objects in the current document. These references are stored in the array in the order in which they are defined in the source code. The **"appletID"** argument is used to access items in the array and this can either be a string containing the applet name as defined within the <APPLET> tags in the HTML source, or an integer (with '0' being the first item in the array).

Both examples below return the same results; the first uses the defined names of the applets and the second uses their reference number within the array.

Code:

```
document.applets["appletname1"]
document.applets["appletname2"]
document.applets["appletname3"]

document.applets[0]
document.applets[1]
document.applets[2]
```

PROPERTY: Document::bgColor

document.bgColor = "colorinfo"

This property defines a document's background color. The **"colorinfo"** argument is a string that can contain either the hexadecimal definition of the color or its literal description. If you use the hex definition of a color it must be in the format rrggbb - for example, the hex value for the named color 'forest green' is '228B22'.

Both lines in the follwing code do exactly the same thing, the first using the hex value of a color and the second using its name.

Code:

```
document.bgColor = "228B22"
document.bgColor = "forestgreen"
```

PROPERTY: Document::cookie

document.cookie[= "expression(s)"]

This property is a string that returns a report detailing all visible and un-expired cookies that are associated with the specified document. The value returned using this method only contains the name and value attributes of the associated cookies in a single string.

When setting the **cookie** property, which can be done at any time, the following syntax must be used:

"name" = "value"; expires = "date"; path = "directory"; domain = "domainName"; secure

The "date" parameter must be in the format as returned by the **toGMTString()** method of the Date object. The expires attribute is optional; not setting this will mean that the cookie will expire when the user shuts down their browser. If set, the cookie survives until the set expiry date. "domainName" sets the cookie's visibility to a particular domain although this attribute is rarely use as it defaults to the domain of the carrying document and visibility of the cookie is normally restricted to this document alone. The path parameter is similar to domain in that it restricts the cookie's visibilty to the specified directory on the web server. The secure attribute is less commonly used. It is a Boolean (true or false) that, when true, suggests that the browser should only make secure (SSL) URL requests when the cookie is sent to the server.

Code:
```
document.write(document.cookie)
```

Output:
```
usr_id=13455; bookmark=products.html
```

OBJECT: Date

new Date()>

new Date(milliseconds)

new Date(dateString)

**new Date(yr_num, mo_num, day_num [, hr_num, min_num,
sec_num, ms_num])**

The **Date** object allows you to work programatically with dates and times. You create a **Date** object using the Date constructor as shown in the syntax above and the available parameters are as follows:

milliseconds - an integer that represents the number of milliseconds since 01/01/70 00:00:00.

dateString - a string that represents the date in a format that is recognized by the **Date.parse** method.

yr_num, mo_num, day_num - an integer that represents the year, month or day of the date.

hr_num, min_num, sec_num, ms_num - an integer that represents the hours, minutes, seconds and milliseconds.

If you don't supply any of the above parameters, JavaScript creates an object for today's date according to the time on the local machine. If any arguments are supplied, you have to include the year, month and day as a minimum, with the time parameters being optional. Note that if you only supply some arguments, any not supplied are set to 0.

All dates are calculated in milliseconds from 01 January, 1970 00:00:00 Universal Time (UTC) with a day containing 86,400,000 milliseconds. The range of a **Date** object relative to 01/01/1970 (UTC) is -100,000,000 to 100,000,000 days and both Universal (UTC) time and Greenwich Mean Time (GMT) are supported.

The following code uses **Date** objects to calculate the time remaining, in days, to the start of the next millennium.

Code:
```
d = new Date()    //today's date
mill=new Date(3000, 00, 01, 00, 00, 00)    //Next millennium start date
diff = mill-d    //difference in milliseconds
mtg = new String(diff/86400000)    //calculate days and convert to string
point=mtg.indexOf(".")    //find the decimal point
days=mtg.substring(0,point)    //get just the whole days
document.write("There are only " + days + " days remaining to the start of the next
millennium.")
```

Output:
```
There are only 365033 days remaining to the start of the next millennium.
```

PROPERTIES

constructor Property
This property returns a reference to the function that created the **Date** object's prototype.

Syntax: **object.constructor**

prototype Property
This property represents the prototype for the object's class and can be used to add properties and methods to all instances of that class.

Syntax: **object.prototype**

METHODS

getDate Method
This method returns an integer (between 1 and 31) representing the day of the month for the specified (local time) date.

Syntax: **object.getDate( )**

getDay Method
This method returns an integer (0 for Sunday thru 6 for Saturday) representing the day of the week.

Syntax: **object.getDay( )**

getFullYear Method
This method returns an integer representing the year of a specified date. The integer returned is a four digit number, 1999, for example, and this method is to be prefered over getYear.

Syntax: **object.getFullYear( )**

getHours Method
This method returns an integer between 0 and 23 that represents the hour (local time) for the specified date.

Syntax: **object.getHours( )**

getMilliseconds Method
This method returns an integer between 0 and 999 that represents the milliseconds (local time) for the specified date.

Syntax: **object.getMilliseconds( )**

getMinutes Method
This method returns an integer between 0 and 59 that represents the minutes (local time) for the specified date.

Syntax: **object.getMinutes( )**

getMonth Method
This method returns an integer (0 for January thru 11 for December) that represents the month for the specified date.

Syntax: **object.getMonth( )**

getSeconds Method
This method returns an integer between 0 and 59 that represents the seconds (local time) for the specified date.

Syntax: **object.getSeconds( )**

getTime Method
This method returns a numeric value representing the number of milliseconds since midnight 01/01/1970 for the specified date.

Syntax: **object.getTime( )**

getTimezoneOffset Method
This method returns the difference in minutes between local time and Greenwich Mean Time. This value is not a constant, as you might think, because of the practice of using Daylight Saving Time.

Syntax: **object.getTimezoneOffset( )**

getUTCDate Method
This method returns an integer between 1 and 31 that represents the day of the month, according to universal time, for the specified date.

Syntax: **object.getUTCDate( )**

getUTCDay Method
This method returns an integer (0 for Sunday thru 6 for Saturday) that represents the day of the week, according to universal time, for the specified date.

Syntax: **object.getUTCDay( )**

getUTCFullYear Method
This method returns a four-digit absolute number that represents the year, according to universal time, for the supplied date.

Syntax: **object.getUTCFullYear( )**

getUTCHours Method
This method returns an integer between 0 and 23 that represents the hours, according to universal time, in the supplied date.

Syntax: **object.getUTCHours( )**

getUTCMilliseconds Method
This method returns an integer between 0 and 999 that represents the milliseconds, according to universal time, in the specified date.

Syntax: **object.getUTCMilliseconds( )**

getUTCMinutes Method
This method returns an integer between 0 and 59 that represents the minutes, in universal time, for the supplied date.

Syntax: **object.getUTCMinutes( )**

getUTCMonth Method
This method returns an integer, 0 for January thru 11 for December, according to universal time, for the specified date.

Syntax: **object.getUTCMonth**

getUTCSeconds Method
This method returns an integer between 0 and 59 that represents the seconds, according to

universal time, for the specified date.

Syntax: **object.getUTCSeconds( )**

parse Method
This method returns takes a date string and returns the number of milliseconds since January 01 1970 00:00:00.

Syntax: **Date.parsedateString**

setDate Method
This method is used to set the day of the month, using an integer from 1 to 31, for the supplied date according to local time.

Syntax: **object.setDate(dateVal)**

setFullYear Method
This method is used to set the full year for the supplied date according to local time.

Syntax: **object.setFullYear(yearVal [, monthVal, dayVal])**

setHours Method
This method is used to set the hours for the supplied date according to local time.

Syntax: **object.setHours(hoursVal [, minutesVal, secondsVal, msVal])**

setMilliseconds Method
This method is used to set the milliseconds for the supplied date according to local time. The **millisecondsVal** parameter expects a number between 0 and 999 athough if this is exceeded, the **setMilliseconds** method will automatically increment other values in the **Date** object, e.g. if 1020 is specified, the seconds value is incremented by one and **millisecondsVal** is set to 20.

Syntax: **object.setMilliseconds(millisecondsVal)**

setMinutes Method
This method is used to set the minutes for the supplied date according to local time.

Syntax: **object.setMinutes(minutesVal [, secondsVal, msVal])**

setMonth Method
This method is used to set the month for the supplied date according to local time.

Syntax: **object.setMonth(monthVal [, dayVal])**

setSeconds Method
This method is used to set the seconds for the specified date according to local time.

Syntax: **object.setSeconds(secondsVal [, msVal)**

setTime Method
This method is used to set the time of a **Date** object according to local time. The **timeVal** argument is an integer that represents the number of milliseconds elapsed since 1 January 1970 00:00:00.

Syntax: **object.setTime(timeVal)**

setUTCDate Method
This method is used to set the day of the month, using an integer from 1 to 31, for the supplied

date according to universal time.

Syntax: **object.setUTCDate(dateVal)**

setUTCFullYear Method
This method is used to set the full year for the supplied date according to universal time.

Syntax: **document.setUTCFullYear(yearVal [, monthVal, dayVal])**

setUTCHours Method
This method is used to set the hours for the supplied date according to universal time.

Syntax: **object.setUTCHours(hoursVal [, minutesVal, secondsVal, msVal])**

setUTCMilliseconds Method
This method is used to set the milliseconds for the supplied date according to universal time. The **millisecondsVal** parameter expects a number between 0 and 999 athough if this is exceeded, the **setMilliseconds** method will automatically increment other values in the **Date** object, e.g. if 1020 is specified, the seconds value is incremented by one and **millisecondsVal** is set to 20.

Syntax: **object.setUTCMilliseconds(millisecondsVal)**

setUTCMinutes Method
This method is used to set the minutes for the supplied date according to universal time.

Syntax: **object.setUTCMinutes(minutesVal [, secondsVal, msVal])**

setUTCMonth Method
This method is used to set the month for the supplied date according to universal time.

Syntax: **object.setUTCMonth(monthVal [, dayVal])**

setUTCSeconds Method
This method is used to set the seconds for the specified date according to universal time.

Syntax: **object.setUTCSeconds(secondsVal [, msVal)**

toGMTString Method
This method converts a local date to Greenwich Mean Time.

Syntax: **object.toGMTString( )**

toLocaleString Method
This method uses the relevant locale's date conventions when converting a date to a string.

Syntax: **object.toLocaleString( )**

toSource Method
This method is used to return the source code that created the specified **Date** object.

Syntax: **object.toSource( )**

toString Method
This method returns a string that represents the **Date** object. This method is automatically called by JavaScript whenever a **Date** object needs to be displayed as text (as with many of the other methods of the **Date** object).

Syntax: **object.toString( )**

toUTCString Method
This method uses the universal time convention when converting a date to a string.

Syntax: **object.toUTCString( )**

UTC Method
This method returns the number of milliseconds from the date in a **Date** object since January 1, 1970 00:00:00 according to universal time. This is a static method of **Date** so the format is always **Date.UTC()** as opposed to **objectName.UTC()**.

Syntax: **Date.UTC(year, month, day [, hours, minutes, seconds, ms])**

valueOf Method
This method is used to return a primitive value, as a number data type, of the specified **Date** object. This is returned as the number of millisecond since January 1, 1970 00:00:00.

Syntax: **object.valueOf( )**

METHOD: Date::parse

Date.parse(dateString)

This method takes a date string an returns the number of milliseconds since January 01 1970 00:00:00, according to local time. It accepts the standard date syntax, e.g. "Friday, 9 July, 11:10:00 GMT+0130". If a time zone is not specified, this method assumes that the supplied string is local time.

This is a static method of **Date** and, subsequently, the syntax is always **Date.parse()** as opposed to **objectName.parse()**.

The following code uses the **Date.parse** method to set the value of the myDate object to that of the supplied string.

Code:
```
myDate = new Date()
myDate.setTime(Date.parse("July 3, 1999"))
document.write(myDate)
```

Output:
```
Sat Jul 3 00:00:00 UTC+0100 1999
```

METHOD: Date::setFullYear

object.setFullYear(yearVal [, monthVal, dayVal])

This method is used to set the full year for the supplied date according to local time. If you do not supply the **monthVal** and **dayVal** arguments, JavaScript will use the values returned using the **getMonth** and **getDay** methods. Also, if the supplied argument is outside the range expected, the **setFullYear** method will alter the other parameters accordingly (see example below).

The available parameters are as follows:

yearVal - this value is an integer representing the year, e.g. 1999

monthVal - an integer representing the month (0 for January thru 11 for December)

dayVal - this value is an integer that represents the day of the month (1 thru 31). If you supply this parameter you also must supply the **monthVal**.

The following code uses the **setFullYear** method to change the value of the myDate object and also demonstrates how this method will adjust the other parameters if a value is supplied that exceeds the expected range. In this case the **dayVal** supplied is 35 which causes the **monthVal** value to be incremented by 2. This is calculated thus:

35 - 31(maximum expected value for month) = 4 (this increments **monthVal** by one).

The result is that **setFullYear** method uses 4 for for the **dayVal** and increments the **monthVal** by one, from 8 to 9.

Code:
```
myDate = new Date()
document.write(myDate +"<br>")
myDate.setFullYear(1999, 08, 35)
document.write(myDate)
```

Output:
```
Fri Jul 9 12:32:32 UTC+0100 1999
Sat Sep 4 12:32:32 UTC+0100 1999
```

METHOD: Date::setHours

object.setHours(hoursVal [, minutesVal, secondsVal, msVal])

This method is used to set the hours for the supplied date according to local time. If you do not supply the **minutesVal** and **secondsVal** and **msVal** arguments, JavaScript will use the values returned using the **getUTCMinutes**, **getUTCSeconds** and **getMilliseconds** methods. Also, if the supplied argument is outside the range expected, the **setHours** method will alter the other parameters accordingly (see example below).

The available parameters are as follows:

hoursVal - this value is an integer (0 thru 23) representing the hour.

minutesVal - an integer (0 thru 59) representing the minutes.

secondsVal - this value is an integer (0 thru 59) that represents the seconds. If you supply this parameter you also must supply **minutesVal**.

msVal - this value is a number between 0 and 999 that represents milliseconds. If this value is supplied, you must also supply **minutesVal** and **secondsVal**.

The following code uses the **setHours** method to change the value of the myDate object and also demonstrates how this method will adjust the other parameters if a value is supplied that exceeds the expected range. In this case the **minutesVal** supplied is 100 which causes the **hourVal** value to be incremented by 1. This is calculated thus:

100 - 60 (maximum expected value for minutes) = 40 (this increments **hoursVal** by one).

The result of this calculation (40) is then used for the **minutesVal** parameter.

Code:
```
myDate = new Date()
document.write(myDate +"<br>")
myDate.setHours(15, 100)
document.write(myDate)
```

Output:
```
Fri Jul 9 13:47:32 UTC+0100 1999
Fri Jul 9 16:40:32 UTC+0100 1999
```

METHOD: Date::setMinutes

---

object.setMinutes(minutesVal [, secondsVal, msVal])

This method is used to set the minutesfield for the supplied date according to local time. If you do not supply the **secondsVal** and **msVal** arguments, JavaScript will use the values returned using the **getUTCSeconds** and **getMilliseconds** methods. Also, if the supplied argument is outside the range expected, the **setMinutes** method will alter the other parameters accordingly (see example below).

The available parameters are as follows:

minutesVal - an integer (0 thru 59) representing the minutes.

secondsVal - this value is an integer (0 thru 59) that represents the seconds. If you supply this parameter you also must supply **minutesVal**.

msVal - this value is a number between 0 and 999 that represents milliseconds. If this value is supplied, you must also supply **minutesVal** and **secondsVal**.

The following code uses the **setMinutes** method to change the value of the myDate object and also demonstrates how this method will adjust the other parameters if a value is supplied that exceeds the expected range. In this case the **secondsVal** supplied is 100 which causes the **minutesVal** value to be incremented by 1. This is calculated thus:

100 - 60 (maximum expected value for seconds) = 40 (this increments **minuteVal** by one).

The result of this calculation (40) is then used for the **secondsVal** parameter.

Code:
```
myDate = new Date()
document.write(myDate +"<br>")
myDate.setMinutes(15, 100)
document.write(myDate)
```

Output:
```
Fri Jul 9 13:47:32 UTC+0100 1999
Fri Jul 9 13:16:40 UTC+0100 1999
```

METHOD: Date::setMonth

object.setMonth(monthVal [, dayVal])

This method is used to set the month for the supplied date according to local time. If you do not supply the **dayVal** argument, JavaScript will use the value returned using the **getDate** method. Also, if the supplied argument is outside the range expected, the **setMonth** method will alter the other parameters of the date object accordingly (see example below).

The available parameters are as follows:

monthVal - an integer (0 for January thru 11 for December) representing the month.

dayVal - this value is an integer (1 thru 31) that represents the day of the month.

The following code uses the **setMonth** method to change the value of the myDate object and also demonstrates how this method will adjust the other parameters if a value is supplied that exceeds the expected range. In this case the **monthVal** supplied is 13 which causes the year to be incremented by 1. This is calculated thus:

13 - 12 (maximum expected value for seconds) = 1 (this increments the year by one).

The result of this calculation (1 = February) is then used for the **monthVal** parameter.

Code:
```
myDate = new Date()
document.write(myDate +"<br>")
myDate.setMonth(13)
document.write(myDate)
```

Output:
```
Fri Jul 9 13:47:32 UTC+0100 1999
Wed Feb 9 13:16:40 UTC+0100 2000
```

METHOD: Date::setSeconds

object.setSeconds(secondsVal [, msVal])

This method is used to set the seconds for the supplied date according to local time. If you do not supply the **msVal** argument, JavaScript will use the value returned using the **getMilliseconds** method. Also, if the supplied argument is outside the range expected, the **setSeconds** method will alter the other parameters of the date object accordingly (see example below).

The available parameters are as follows:

secondsVal - an integer (0 thru 59) representing the seconds.

msVal - this value is an integer (1 thru 999) that represents the milliseconds.

The following code uses the **setSeconds** method to change the value of the myDate object and also demonstrates how this method will adjust the other parameters if a value is supplied that exceeds the expected range. In this case the **secondsVal** supplied is 90 which causes the minutes to be incremented by 1. This is calculated thus:

90 - 60 (maximum expected value for seconds) = 30 (this increments the minutes by one).

The result of this calculation (30) is then used for the **secondsVal** parameter.

Code:
```
myDate = new Date()
document.write(myDate +"<br>")
myDate.setSeconds(90)
document.write(myDate)
```

Output:
```
Fri Jul 9 13:30:20 UTC+0100 1999
Fri Jul 9 13:31:30 UTC+0100 1999
```

METHOD: Date::setUTCFullYear

object.setUTCFullYear(yearVal [, monthVal, dayVal])

This method is used to set the full year for the supplied date according to universal time. If you do not supply the **monthVal** and **dayVal** arguments, JavaScript will use the values returned using the **getMonth** and **getDay** methods. Also, if the supplied argument is outside the range expected, the **setUTCFullYear** method will alter the other parameters accordingly (see example below).

The available parameters are as follows:

yearVal - this value is an integer representing the year, e.g. 1999

monthVal - an integer representing the month (0 for January thru 11 for December)

dayVal - this value is an integer that represents the day of the month (1 thru 31). If you supply this parameter you also must supply the **monthVal**.
The following code uses the **setUTCFullYear** method to change the value of the myDate object and also demonstrates how this method will adjust the other parameters if a value is supplied that exceeds the expected range. In this case the **dayVal** supplied is 35 which causes the **monthVal** value to be incremented by 2. This is calculated thus:

35 - 31(maximum expected value for month) = 4 (this increments **monthVal** by one).

The result is that **setUTCFullYear** method uses 4 for for the **dayVal** and increments the **monthVal** by one, from 8 to 9.

Code:
```
myDate = new Date()
document.write(myDate +"<br>")
myDate.setUTCFullYear(1999, 08, 35)
document.write(myDate)
```

Output:
```
Fri Jul 9 12:32:32 UTC+0100 1999
Sat Sep 4 12:32:32 UTC+0100 1999
```

METHOD: Date::setUTCHours

object.setUTCHours(hoursVal [, minutesVal, secondsVal, msVal])

This method is used to set the hours for the supplied date according to local time. If you do not supply the **minutesVal** and **secondsVal** and **msVal** arguments, JavaScript will use the values returned using the **getUTCMinutes**, **getUTCSeconds** and **getMilliseconds** methods. Also, if the supplied argument is outside the range expected, the **setUTCHours** method will alter the other parameters accordingly (see example below).

The available parameters are as follows:

hoursVal - this value is an integer (0 thru 23) representing the hour.

minutesVal - an integer (0 thru 59) representing the minutes.

secondsVal - this value is an integer (0 thru 59) that represents the seconds. If you supply this parameter you also must supply **minutesVal**.

msVal - this value is a number between 0 and 999 that represents milliseconds. If this value is supplied, you must also supply **minutesVal** and **secondsVal**.

The following code uses the **setUTCHours** method to change the value of the myDate object and also demonstrates how this method will adjust the other parameters if a value is supplied that exceeds the expected range. In this case the **minutesVal** supplied is 100 which causes the **hourVal** value to be incremented by 1. This is calculated thus:

100 - 60 (maximum expected value for minutes) = 40 (this increments **hoursVal** by one).

The result of this calculation (40) is then used for the **minutesVal** parameter.

Code:
```
myDate = new Date()
document.write(myDate +"<br>")
myDate.setUTCHours(15, 100)
document.write(myDate)
```

Output:
```
Fri Jul 9 13:47:32 UTC+0100 1999
Fri Jul 9 16:40:32 UTC+0100 1999
```

METHOD: Date::setUTCMinutes

object.setUTCMinutes(minutesVal [, secondsVal, msVal])

This method is used to set the minutes field for the supplied date according to universal time. If you do not supply the **secondsVal** and **msVal** arguments, JavaScript will use the values returned using the **getUTCSeconds** and **getMilliseconds** methods. Also, if the supplied argument is outside the range expected, the **setUTCMinutes** method will alter the other parameters accordingly (see example below).

The available parameters are as follows:

minutesVal - an integer (0 thru 59) representing the minutes.

secondsVal - this value is an integer (0 thru 59) that represents the seconds. If you supply this parameter you also must supply **minutesVal**.

msVal - this value is a number between 0 and 999 that represents milliseconds. If this value is supplied, you must also supply **minutesVal** and **secondsVal**.

The following code uses the **setUTCMinutes** method to change the value of the myDate object and also demonstrates how this method will adjust the other parameters if a value is supplied that exceeds the expected range. In this case the **secondsVal** supplied is 100 which causes the **minutesVal** value to be incremented by 1. This is calculated thus:

100 - 60 (maximum expected value for seconds) = 40 (this increments **minuteVal** by one).

The result of this calculation (40) is then used for the **secondsVal** parameter.

Code:
```
myDate = new Date()
document.write(myDate +"<br>")
myDate.setUTCMinutes(15, 100)
document.write(myDate)
```

Output:
```
Fri Jul 9 13:47:32 UTC+0100 1999
Fri Jul 9 13:16:40 UTC+0100 1999
```

METHOD: Date::setUTCMonth

object.setUTCMonth(monthVal [, dayVal])

This method is used to set the month for the supplied date according to universal time. If you do not supply the **dayVal** argument, JavaScript will use the value returned using the **getDate** method. Also, if the supplied argument is outside the range expected, the **setUTCMonth** method will alter the other parameters of the date object accordingly (see example below).

The available parameters are as follows:

monthVal - an integer (0 for January thru 11 for December) representing the month.

dayVal - this value is an integer (1 thru 31) that represents the day of the month.

The following code uses the **setUTCMonth** method to change the value of the myDate object and also demonstrates how this method will adjust the other parameters if a value is supplied that exceeds the expected range. In this case the **monthVal** supplied is 13 which causes the year to be incremented by 1. This is calculated thus:

13 - 12 (maximum expected value for seconds) = 1 (this increments the year by one).

The result of this calculation (1 = February) is then used for the **monthVal** parameter.

Code:
```
myDate = new Date()
document.write(myDate +"<br>")
myDate.setUTCMonth(13)
document.write(myDate)
```

Output:
```
Fri Jul 9 13:47:32 UTC+0100 1999
Wed Feb 9 13:16:40 UTC+0100 2000
```

METHOD: Date::setUTCSeconds

object.setUTCSeconds(secondsVal [, msVal])

This method is used to set the seconds for the supplied date according to universal time. If you do not supply the **msVal** argument, JavaScript will use the value returned using the **getMilliseconds** method. Also, if the supplied argument is outside the range expected, the **setUTCSeconds** method will alter the other parameters of the date object accordingly (see example below).

The available parameters are as follows:

secondsVal - an integer (0 thru 59) representing the seconds.

msVal - this value is an integer (1 thru 999) that represents the milliseconds.

The following code uses the **setUTCSeconds** method to change the value of the myDate object and also demonstrates how this method will adjust the other parameters if a value is supplied that exceeds the expected range. In this case the **secondsVal** supplied is 90 which causes the minutes to be incremented by 1. This is calculated thus:

90 - 60 (maximum expected value for seconds) = 30 (this increments the minutes by one).

The result of this calculation (30) is then used for the **secondsVal** parameter.

Code:
```
myDate = new Date()
document.write(myDate +"<br>")
myDate.setUTCSeconds(90)
document.write(myDate)
```

Output:
```
Fri Jul 9 13:30:20 UTC+0100 1999
Fri Jul 9 13:31:30 UTC+0100 1999
```

METHOD: Date::toGMTString

object.toGMTString( )

This method converts a local date to Greenwich Mean Time.

The following code uses the **toGMTString** method to convert the date set in the myDate object to Greenwich Mean Time and returns this date as a string.

Code:
```
myDate = new Date()
document.write(myDate.toGMTString())
```

Output:
```
Fri, 29 Oct 1999 16:28:58 UTC
```

METHOD: Date::toLocaleString

object.toLocaleString( )

This method uses the relevant locale's date conventions when converting a date to a string. This is done by using the default date format of the user's operating system. This takes in to account the differences between date formatting methods used in different countries, e.g. in the U.K. the date comes before the month as opposed to the U.S. convention of the month first.

The following code uses the **toLocaleString** method to convert the date set in the myDate object to a string. For the purposes of this example, the locale of the operating system is set to that of the United Kingdom.

Code:
```
myDate = new Date(99, 04, 23)
document.write(myDate.toLocaleString())
```

Output:
```
23/05/1999 00:00:00
```

PROPERTY: Document::domain

document.domain = "domaininfo"

This property sets or returns the domain name of the server from which the document originated. This defaults to the domain name of the server that the document was retreived from, but can be changed to a suffix (and only a suffix) of this name. This allows the sharing of script properties, security allowing, between documents delivered from different servers providing they share the same domain suffix.

The way you can alter the domain name property is very limited. For example, if a document was retreived from the URL 'search.devguru.com', you could change the domain property to 'devguru.com' but not 'search.devguru'.

These examples relate to a document retreived from the URL 'search.devguru.com'

Code:
```
document.domain = "devguru.com"     // This example is o.k.

document.domain = "search.devguru"     // This example is not allowed

document.domain = "devguru.net"     // This example is not allowed
```

PROPERTY: Document::embeds

document.embeds["embed_objID"]

This property is an array containing references to all the embedded objects in the current document. These references are stored in the array in the order in which they are defined in the source code. The **"embed_objID"** argument is used to access items in the array and this can either be a string containing the embedded object's name as defined within the <EMBED> tag in the HTML source, or an integer (with '0' being the first item in the array).

Both examples below return the same results; the first uses the defined names of the embedded objects and the second uses their reference number within the array.

Code:
```
document.embeds["embed_obj1"]
document.embeds["embed_obj2"]
document.embeds["embed_obj3"]

document.embeds[0]
document.embeds[1]
document.embeds[2]
```

PROPERTY: Document::fgColor

document.fgColor = "colorinfo"

This property defines a document's foreground (text) color. The **"colorinfo"** argument is a string that can contain either the hexadecimal definition of the color or its literal description. If you use the hex definition of a color it must be in the format rrggbb - for example, the hex value for the named color 'forest green' is '228B22'.

Both lines in the follwing code do exactly the same thing, the first using the hex value of a color and the second using its name.

Code:

```
document.fgColor = "228B22"
document.fgColor = "forestgreen"
```

PROPERTY: Document::forms

---

document.forms["formID"]

This property is an array containing references to all the **Form** objects in the current document. These references are stored in the array in the order in which they are defined in the source code. The **"formID"** argument is used to access items in the array and this can either be a string containing the form name as defined within the <FORM> tag in the HTML source, or an integer (with '0' being the first item in the array).

All examples below return the same results; the first uses the defined names of the forms and the second uses their reference number within the array.

Code:

```
document.forms["formname1"]
document.forms["formname2"]
document.forms["formname3"]

document.forms[0]
document.forms[1]
document.forms[2]
```

Because a separate property is assigned to the document object for each form, (see the document.formname property) you can also use the following syntax.

```
document.formname1
document.formname2
document.formname3
```

PROPERTY: Document::images

document.images["imageID"]

This property is an array containing references to all the **Image** objects in the current document. These references are stored in the array in the order in which they are defined in the source code. The **"imageID"** argument is used to access items in the array and this can either be a string containing the image name as defined within the <IMG> tag in the HTML source, or an integer (with '0' being the first item in the array).

Both examples below return the same results; the first uses the defined names of the images and the second uses their reference number within the array.

Code:
```
document.images["imagename1"]
document.images["imagename2"]
document.images["imagename3"]

document.images[0]
document.images[1]
document.images[2]
```

PROPERTY: Document::lastModified

document.lastModified

This property returns a string relating to the date that the document was last modified, which is usually, but not always, contained in the HTTP header of a document. When this data is supplied, the server from which the document originated interogates the file for its 'last modified' date and includes this in the header information of the document. If a particular server doesn't do this, and no 'date last modified' data exists in the HTTP header, JavaScript will return a value of '0', which it interprets as 'Janurary 1, 1970 GMT'.

The following code gets the last modified date of a document and displays it in the browser.

Code:
```
datelastmod = document.lastModified
document.write("This document was last modified on " + datelastmod)
```

Output:
```
"This document was last modified on 10/28/97 12/06/56"
```

PROPERTY: Document::layers

document.layers["layerID"]

This property is an array containing references to all the **Layer** objects in the current document. These references are stored in the array in the order in which they are defined in the source code. The **"layerID"** argument is used to access items in the array and this can either be a string containing the layer name as defined within the <LAYER> or <ILAYER> tag in the HTML source, or an integer (with '0' being the first item in the array).

Both examples below return the same results; the first uses the defined names of the layers and the second uses their reference number within the array.

Note that when accessing the layers by their reference integer, as opposed to name, they are stored in the array in z-order (from back to front, with the back-most layer indexed as '0').

Code:

```
document.layers["layername1"]
document.layers["layername2"]
document.layers["layername3"]

document.layers[0]
document.layers[1]
document.layers[2]
```

PROPERTY: Document::linkColor

document.linkColor = "colorinfo"

This property defines the color of any hyperlinks in the document. The **"colorinfo"** argument is a string that can contain either the hexadecimal definition of the color or its literal description. If you use the hex definition of a color it must be in the format rrggbb - for example, the hex value for the named color 'forest green' is '228B22'.

Both lines in the follwing code do exactly the same thing, the first using the hex value of a color and the second using its name.

Code:
```
document.linkColor = "228B22"
document.linkColor = "forestgreen"
```

PROPERTY: Document::links

---

document.links["linkID"]

This property is an array containing references to all the **Area** and **Link** objects in the current document. These references are stored in the array in the order in which they are defined in the source code. The **"linkID"** argument is an integer relating to a link defined within a <A HREF = " "> or <AREA HREF = " "> tag in the HTML source.

Code:

```
document.links[0]
document.links[1]
document.links[2]
```

PROPERTY: Document::plugins

document.plugins["pluginID"]

This property is an array containing references to all the **Plugin** objects in the current document. These references are stored in the array in the order in which they are defined in the source code, and are accessed using the **"pluginID"** argument, which is an integer with the first plugin object being '0'.

Code:
```
document.plugins[0]
document.plugins[1]
document.plugins[2]
```

PROPERTY: Document::vlinkColor

document.vlinkColor = "colorinfo"

This property defines the color of any visited links in the document. The **"colorinfo"** argument is a string that can contain either the hexadecimal definition of the color or its literal description. If you use the hex definition of a color it must be in the format rrggbb - for example, the hex value for the named color 'forest green' is '228B22'.

Both lines in the follwing code do exactly the same thing, the first using the hex value of a color and the second using its name.

Code:

```
document.linkColor = "228B22"
document.linkColor = "forestgreen"
```

METHOD: Document::close

document.close( )

This method closes an output stream previously opened with the **document.open** method and forces data collected from any instances of the **document.write** or **document.writeln** methods to be displayed.

The following code demonstrates this and displays the output stream in a new window.

Code:
```
function newWindowTest() {
  var message1 = "Hello, world!"
  var message2 = "This is a test."
  newWindow.document.open("text/html", "replace")
  newWindow.document.writeln("message1)
  newWindow.document.write("message2)
  newWindow.document.close()
}

newWindow=window.open('','','toolbar=no,scrollbars=no,width=200,height=150')
newWindowTest()
```

Output: (to new window)
```
"Hello, world! This is a test."
```

METHOD: Document::getSelection

document.getSelection( )

This method can be used to return a string containing any selected text in the current document.

The following code, when placed in a form in the current document, will display any selected text in a message box when the button is clicked.

Code:
```
<INPUT TYPE="BUTTON" NAME="selectString" VALUE="Show
any highlighted text" onClick="alert('The following text is
selected:\n'+document.getSelection());">
```

METHOD: Document::open

document.open([mimeType[, replace]])

This method is used to open a stream to collect the output from any **write** or **writeln** methods. The first of the optional parameters is **mimeType** which determines the type of document you are writing to; if this parameter is not used, the default value is "text/html". The second parameter is **replace**, also optional, which causes the history entry for the new document to inherit the history entry from the document from which it was opened.

The following code demonstrates this method and displays the output stream in a new window.

Code:
```
function newWindowTest() {
  var message1 = "Hello, world!"
  var message2 = "This is a test."
  newWindow.document.open("text/html", "replace")
  newWindow.document.writeln("message1)
  newWindow.document.write("message2)
  newWindow.document.close()
}

newWindow=window.open('','','toolbar=no,scrollbars=no,width=200,height=150')
newWindowTest()
```

Output: (to new window)
```
"Hello, world! This is a test."
```

METHOD: Document::write

document.write("expression1", [expression2, [...]])

This method is used to write HTML expressions and JavaScript code to the specified document in a current or new window.

Multiple arguments, ("expression1", [expression2, [...]]), can be listed and they will be appended to the document in order of occurrence. They can be of any type supported by JavaScript (string, numeric, logical), but all non-string expressions will be converted to a string before being appended.

In general, it is not necessary to open the document using the **document.open** method, since the **document.write** method will automatically open the file and discard (erase) the contents. However, after the write is complete, you need to close the document by using the **document.close** method. In some browsers, the results of the write may not be completely displayed, due to buffering, until the close occurs.

Code:
```
newWindow = window.open('', 'newWin')
var tagBoldOpen = "<b>"
var tagBoldClose = "</b>"
newWindow.document.write(tagBoldOpen)
newWindow.document.write("This is some bold text.", tagBoldClose)
newWindow.document.close()
```

EVENT HANDLER: onClick

onClick = myJavaScriptCode

Event handler for **Button**, **Document**, **Checkbox**, **Link**, **Radio**, **Reset**, **Submit**.

The **onClick** event handler executes the specified JavaScript code or function on the occurance of a click event. When used with checkboxes, links, radio, reset and submit buttons, onClick can return a false value which cancels the normal action associated with the click event. With form objects that have default actions this works as follows:

Radio buttons and checkboxes - nothing is set
Submit buttons - submission of the form is cancelled.
Reset buttons - resetting of the form is cancelled.

The **onClick** event handler uses the following **Event** object properties.

type - this property indicates the type of event.
target - this property indicates the object to which the event was originally sent.
***layerX** - the cursor location when the click event occurs.
***layerY** - the cursor location when the click event occurs.
***pageX** - the cursor location when the click event occurs.
***pageY** - the cursor location when the click event occurs.
***screenX** - the cursor location when the click event occurs.
***screenY** - the cursor location when the click event occurs.
which - 1 represents a left mouse click and 3 a right click.
modifiers - lists the modifier keys (shift, alt, ctrl, etc.) held down when the click event occurs.

*when a link is clicked.

The following example shows the use of the **onClick** event handler to offer users the chance to cancel the resetting of a from when clicking on a Reset button.

Code:
```
<INPUT TYPE="RESET" onClick="return confirm('Are you sure?')">
```

OBJECT: Button

A **Button** object is created with every instance of an HTML <INPUT> tag (with a 'type' value set as 'button') on a form. These objects are then stored in the elements array of the parent form and accessed using either the name defined within the HTML tag or an integer (with '0' being the first element defined, in source order, in the specified form).

PROPERTIES

form Property
This property returns a reference to the button's parent **form**.

Syntax: **button.form**

name Property
This property sets or returns the value of the button's **name** attribute.

Syntax: **button.name**

type Property
Every element on a form has an associated **type** property. In the case of a **Button** object, the value of this property is always "button".

Syntax: **button.type**

value Property
This property sets or returns the button's **value** attribute. This is the text that is actually displayed on the button face.

Syntax: **button.value**

METHODS

blur Method
This method removes the **focus** from the specified **Button** object.

Syntax: **button.blur( )**

click Method
This method simulates a mouse-click on the button.

Syntax: **button.click( )**

focus Method
This method gives focus to the specified **Button** object.

Syntax: **button.focus( )**

handleEvent Method
This method calls the handler for the specified event.

Syntax: **button.handleEvent(event)**

EVENT HANDLERS

onBlur Event handler

This event handler executes some specified JavaScript code on the occurrence of a **blur** event (when the button object loses focus).

Syntax: **button.onBlur="myJavaScriptCode"**

onClick Event handler

This event handler executes some specified JavaScript code on the occurrence of a **click** event (when the button object is clicked).

Syntax: **button.onClick="myJavaScriptCode"**

onFocus Event handler

This event handler executes some specified JavaScript code on the occurrence of a **focus** event (when the button object receives focus).

Syntax: **button.onFocus="myJavaScriptCode"**

onMouseDown Event handler

This event handler executes some specified JavaScript code on the occurrence of an **onMouseDown** event (when a mouse button is depressed).

Syntax: **button.onMouseDown="myJavaScriptCode"**

onMouseUp Event handler

This event handler executes some specified JavaScript code on the occurrence of an **onMouseUp** event (when a mouse button is released).

Syntax: **button.onMouseUp="myJavaScriptCode"**

PROPERTY: Button::form

button.form

This property returns a reference to the **Button** object's parent **Form**.

The following code dispays the name of the **Button** object's parent **Form** when it is clicked and assumes, for the purposes of this example, that the **Form** is called "myForm".

Code:

```
<INPUT NAME="myButton" TYPE="button" VALUE="Form name?" onClick= "document.write
(document.myForm.myButton.form.name)">
```

PROPERTY: Button::value

button.value

This property sets or returns the button's **Value** attribute. This is the text that is actually displayed on the **Button** face.

The following code uses the button's **onClick** event handler to call a JavaScript function that changes the **Value** attribute of the **Button**..

Code:

```
<form name="myForm" title="myForm">
<INPUT NAME="myButton" TYPE="button" VALUE="Click to change value"
onClick=valChange()>

<script language="javascript">
function valChange()      {
    document.myForm.myButton.value="Value has changed"
}
</script>
</form>
```

EVENT HANDLER: onBlur

onBlur = myJavaScriptCode

Event handler for **Button**, **Checkbox**, **FileUpload**, **Layer**, **Password**, **Radio**, **Reset**, **Select**, **Submit**, **Text**, **TextArea**, **Window**.

The **onBlur** event handler executes the specified JavaScript code or function on the occurance of a blur event. This is when a window, frame or form element loses focus. This can be caused by the user clicking outside of the current window, frame or form element, by using the TAB key to cycle through the various elements on screen, or by a call to the **window.blur** method.

The **onBlur** event handler uses the following **Event** object properties.

type - this property indicates the type of event.
target - this property indicates the object to which the event was originally sent.

The following example shows the use of the **onBlur** event handler to ask the user to check that the details given are correct. Note that the first line is HTML code.

Code:

```
Enter email address <INPUT TYPE="text" VALUE="" NAME="userEmail" onBlur=addCheck()>

<script type="text/javascript" language="JavaScript">

function addCheck() {
    alert("Please check your email details are correct before submitting")
}

</script>
```

OBJECT: Checkbox

A **Checkbox** object is created with every instance of an HTML <INPUT> tag (with a 'type' value set as 'checkbox') on a **Form**. These objects are then stored in the **Elements** array of the parent form and accessed using either the name defined within the HTML tag or an integer (with '0' being the first element defined, in source order, in the specified **Form**).

PROPERTIES

checked Property
This property is a boolean value that sets or returns the current state of the **Checkbox** object; true if checked and false otherwise.

Syntax: **checkbox.checked**

defaultChecked Property
This property sets or returns the default value of the **checked** property.

Syntax: **checkbox.defaultChecked**

form Property
This property returns a reference to the **Checkbox** object's parent **Form**.

Syntax: **checkbox.form**

name Property
This property sets or returns the **Checkbox** object's **Name** attribute.

Syntax: **checkbox.name**

type Property
Every element on a form has an associated **Type** property. In the case of a **Checkbox** object, the value of this property is always "checkbox".

Syntax: **checkbox.type**

value Property
This property sets the **Value** that is returned when the **Checkbox** is checked.

Syntax: **checkbox.value**

METHODS

blur Method
This method removes the **Focus** from the specified **Checkbox** object.

Syntax: **checkbox.blur( )**

click Method
This method simulates a mouse-click on the **Checkbox** object.

Syntax: **checkbox.click( )**

focus Method
This method gives **Focus** to the specified **Checkbox** object.

Syntax: **checkbox.focus( )**

handleEvent Method
This method calls the handler for the specified **Event**.

Syntax: **checkbox.handleEvent(event)**

EVENT HANDLERS

onBlur Event handler
This event handler executes some specified JavaScript code on the occurrence of a **Blur** event (when the **Checkbox** object loses focus).

Syntax: **checkbox.onBlur="myJavaScriptCode"**

onClick Event handler
This event handler executes some specified JavaScript code on the occurrence of a **Click** event (when the **Checkbox** object is clicked).

Syntax: **checkbox.onClick="myJavaScriptCode"**

onFocus Event handler
This event handler executes some specified JavaScript code on the occurrence of a **Focus** event (when the **Checkbox** object receives focus).

Syntax: **checkbox.onFocus="myJavaScriptCode"**

EVENT HANDLER: onFocus

onFocus = myJavaScriptCode

Event handler for **Button**, **Checkbox**, **FileUpload**, **Layer**, **Password**, **Radio**, **Reset**, **Select**, **Submit**, **Text**, **TextArea**, **Window**.

The **onFocus** event handler executes the specified JavaScript code or function on the occurance of a focus event. This is when a window, frame or form element is given focus. This can be caused by the user clicking on the current window, frame or form element, by using the TAB key to cycle through the various elements on screen, or by a call to the **window.focus** method. Be aware that assigning an alert box to an object's **onFocus** event handler with result in recurrent alerts as pressing the 'o.k.' button in the alert box will return focus to the calling element or object.

The **onFocus** event handler uses the following **Event** object properties.

type - this property indicates the type of event.
target - this property indicates the object to which the event was originally sent.

The following example shows the use of the **onFocus** event handler to replace the default string displayed in the text box. Note that the first line is HTML code and it is accepted that the text box resides on a form called 'myForm'.

Code:
```
<input type="text" name="myText" value="Give me focus" onFocus = "changeVal()">

<script type="text/javascript" language="JavaScript">

s1 = new String(myForm.myText.value)

function changeVal() {
   s1 = "I'm feeling focused"
   document.myForm.myText.value = s1.toUpperCase()
}

</script>
```

OBJECT: FileUpload

A **FileUpload** object provides a field in which the user can enter a file name to be uploaded and used as input and is created with every instance of an HTML <INPUT> tag (with the 'type' attribute set to 'file') on a form. These objects are then stored in the elements array of the parent form and accessed using either the name defined within the HTML tag or an integer (with '0' being the first element defined, in source order, in the specified form).

PROPERTIES

form Property
This property returns a reference to the parent **Form** of the **FileUpload** object.

Syntax: **object.form**

name Property
This property sets or returns the value of the **FileUpoad** object's **name** attribute.

Syntax: **object.name**

type Property
Every element on a form has an associated **type** property. In the case of a **FileUpload** object, the value of this property is always 'file'.

Syntax: **object.type**

value Property
This property sets or returns the **FileUpload** object's **value** attribute. This is a string input by the user that specifies the file to upload.

Syntax: **object.value**

METHODS

blur Method
This method removes the **focus** from the specified **FileUpload** object.

Syntax: **object.blur( )**

click Method
This method simulates a mouse-click on the **FileUpload** object.

Syntax: **object.click( )**

focus Method
This method gives focus to the specified **FileUpload** object.

Syntax: **object.focus( )**

handleEvent Method
This method calls the handler for the specified event.

Syntax: **object.handleEvent(event)**

EVENT HANDLERS

onBlur Event handler

This event handler executes some specified JavaScript code on the occurrence of a **blur** event (when the **FileUpload** object loses focus).

Syntax: **object.onBlur="myJavaScriptCode"**

onChange Event handler

This event handler executes some specified JavaScript code on the occurrence of a **click** event (when the **FileUpload** object loses focus and its value has altered).

Syntax: **object.onClick="myJavaScriptCode"**

onFocus Event handler

This event handler executes some specified JavaScript code on the occurrence of a **focus** event (when the **FileUpload** object receives focus).

Syntax: **object.onFocus="myJavaScriptCode"**

EVENT HANDLER: onChange

onChange = myJavaScriptCode

Event handler for **FileUpload**, **Select**, **Text**, **TextArea**.

The **onChange** event handler executes the specified JavaScript code or function on the occurance of a change event. This is when the data in one of the above form elements is altered by the user. This is used frequently to validate the data that has been entered by the user by calling a specified JavaScript function.

The **onChange** event handler uses the following **Event** object properties.

type - this property indicates the type of event.
target - this property indicates the object to which the event was originally sent.

The following example shows the use of the **onChange** event handler to call a JavaScript function that validates the data input by the user (in this case, the function simply checks wether the entered email address contains the '@' character and displays a relevant message). Note that the first line is HTML code and assumes that the text box is on a form called 'myForm'.

Code:
```
<INPUT TYPE="text" VALUE="Enter email address" NAME="userEmail"
onChange=validateInput(this.value)>

<script type="text/javascript" language="JavaScript">

this.myForm.userEmail.focus()
this.myForm.userEmail.select()

function validateInput() {
userInput = new String()
userInput = this.myForm.userEmail.value
 if (userInput.match("@"))
    alert("Thanks for your interest.")
 else
    alert("Please check your email details are correct before submitting")
}

</script>
```

OBJECT: Select

The **Select** object represents a selection list in a **Form** object. As such, it must be declared inside <FORM> tags. The JavaScript runtime engine creates such an object for every selection list in a particular form, and assigns it to the Form's **elements** array. It is through this array that a particular **Select** object can be accessed, either by index-number or by its NAME attribute.

PROPERTIES

form Property
This property is a reference to the parent form to which a particular **Select** object belongs.

Syntax: **object.form**

length Property
This property contains the number of options in the selection list. For example, to refer to the length of the first **Select** object in the first form of the current document, you could use the following code:

Code:
document.forms[0].elements[0].length

Syntax: **object.length**

name Property
This property consists of a string which gives the name of the selection. For example, to refer to the name of the first **Select** object in the first form of the current document, you could use the following code:

Code:
document.forms[0].elements[0].name

Syntax: **object.name**

options Property
This property is an array of all the options in a particular **Select** object. There is one element (numbered in ascending order from zero) for each <OPTION> tag.

Syntax: **object.options**

selectedIndex Property
This property, which is tainted by default, is an integer relating to the currently-selected option of a **Select** object. If, however, the **Select** object allows for multiple selections (i.e. when the <SELECT> tag includes the MULTIPLE attribute), the **selectedIndex** property will only return the index of the first option selected. For example, the following code would return the index of the selected option of a **Select** object called MySelect in MyForm:

Code:
document.myForm.mySelect.selectedIndex

Syntax: **object.selectedIndex**

type Property

This property holds the type of the **Select** object, having the value "select-one" where only one option can be selected and "select-multiple" where multiple selections are possible. The following code could be used to determine the type of the first **Select** object of the first form of the current document:

Code:
```
document.forms[0].elements[0].type
```

Syntax: **object.type**

METHODS

blur Method
This method removes focus from a selection list.

Syntax: **object.blur()**

focus Method
This method moves the focus to the specified selection list allowing the user to then select from it.

Syntax: **object.focus()**

handleEvent Method
This method calls the handler for a specified event.

Syntax: **object.handleEvent(event)**

NOTE:

The **Select** object also inherits the **watch** and **unwatch** methods from the **Object** object.

EVENT HANDLERS

onBlur EventHandler
This event handler causes JavaScript code to be executed whenever a blur event occurs; i.e. whenever a window, frame or form element loses focus.

Syntax: **onBlur = "myJavaScriptCode"**

onChange EventHandler
This event handler executes JavaScript code whenever a **Select**, **Text** or **Textarea** field loses focus after having been altered.

Syntax: **onChange = "myJavaScriptCode"**

onFocus EventHandler
This event handler executes JavaScript whenever a focus event occurs; i.e. whenever the user focuse on a window, frame or frameset, or inputs to a form element.

Syntax: **onFocus = "myJavaScriptCode"**

PROPERTY: Select::options

object.options

The **options** property is an array of all the options in a particular **Select** object. There is one element (numbered in ascending order from zero) for each <OPTION> tag. You can use the **length** property of this array to refer to the number of options in a particular **Select** object as follows:

Code:
```
document.myForm.mySelect.options.length
```

...or by simply using the **length** property of the **Select** object:

Code:
```
document.myForm.mySelect.length
```

You can use the **options** array to add options to, or delete options from any **Select** object. When adding or altering an option, you assign an **Option** object to a particular element of the array. The following code first creates an **Option** object called Folk, and then assigns it to element # 3 of the **options** array of a user-defined **Select** object called MusicType. If there is a value already assigned to element 3, it will be replaced by Folk, otherwise it will be created along with undefined elements at every index between the one created and the last existant one.

Code:
```
document.forms[0].musicType.options[3] = new Option("Folk", "folk", false, false)
```

Similarly, you can delete any option by assigning the value **null** to the appropriate element of the **options** array:

Code:
```
document.forms[0].musicType.options[2] = null
```

This will have the effect of removing element # 2 from the **options** array, and at the same time compressing the array so that element # 3 becomes # 2, element # 4 becomes # 3 etc. After deleting an option you must refresh the document by using **history.go(0)** at the end of the code. To determine which option of a **Select** object is currently selected, you can use the **selectedIndex** property along with the **options** property as follows:

Code:
```
document.forms[0].musicType.options.selectedIndex
```

...or you could simply use the **selectedIndex** property of the **Select** object:

Code:
```
document.forms[0].musicType.selectedIndex
```

Both the above lines of code return the number of the currently-selected index. If, however, the **Select** object allows for multiple selections (i.e. when the <SELECT> tag includes the MULTIPLE attribute), the **selectedIndex** property will only return the index of the first option selected. To determine all of the selected options, you would have to loop through the **options** array testing each option individually.

OBJECT: Option

new Option([text[, value[, defaultSelected[, selected]]]])

An **Option** object is created for every option in a selection list, and is put in the **options** property of the **Select** object. It can be created in one of two ways: you can either use the HTML <OPTION> tag, or use the **Option** constructor. Using HTML you could create a 'Dachshund' option for a selection list of dog breeds as follows:

Code:
```
<option> Dachshund
```

You could also create the same option using the **Option** constructor and assigning it to an index of the **options** property of the relevent **Select** object:

Code:
```
document.myForm.dogBreed[4] = new Option("Dachshund")
```

After creating an **Option** object in this way you must refresh the document by using **history.go(0)** at the end of the code. Using the **Option** constructor, you can optionally specify a value to be returned to the server when an option is selected and the form submitted (in this case "dachs"):

Code:
```
new Option("Dachshund", "dachs")
```

It is also possible to designate the option to be the pre-selected default selection in the option box display (i.e., this option has the HTML "selected" attribute included inside the "option" tag). This is done by setting the **defaultSelected** argument to be **true**.

Code:
```
new Option("Dachshund", "dachs", true)
```

The **selected** argument is used for multiple selections.

PROPERTIES

defaultSelected Property

This property, by default tainted, is a Boolean value which initially reflects whether an option was declared with the HTML SELECTED attribute, reading **true** if it was and **false** if not

Syntax: **object.defaultSelected**

selected Property

This property, which is tainted by default, is a Boolean value reflecting whether a particular option is selected, returning **true** if it is and **false** if not. The **selected** property can be set at any time, immediately updating the display of the **Select** object.

Syntax: **object.selected**

text Property

This property, by default tainted, reflects the text value following any particular HTML OPTION tag for a **Select** object. It can be reset at any time, immediately updating the display of the selection.

Syntax: **object.text**

value Property
This property, tainted by default, is a string value that is returned to the server when an option is selected and the form submitted. It reflects the VALUE attribute in the HTML. If there is no VALUE attribute, then the **value** property is an empty string.

Syntax: **object.value**

METHODS

The **Select** object inherits the **watch** and **unwatch** methods from the **Object** object.

PROPERTY: Option::defaultSelected

object.defaultSelected

This property, by default tainted, is a Boolean value which initially reflects whether an option was declared with the HTML SELECTED attribute, reading **true** if it was and **false** if not. You can set the **defaultselected** property at any time, and this will override the initial setting, but it won't be reflected in the display of the **Select** object. If, however, you set the **defaultSelected** property of a **Select** object created with the MULTIPLE attribute, any initial defaults are unaffected.

For example, assuming a **Select** object called DogBreed with the first option initially selected by default, the following code would alter that default to the fourth option:

Code:
```
document.myForm.dogBreed.options[3].defaultSelected = true
```

You could then test to see that this has actually happened as follows:

Code:
```
if(document.myForm.dogType.options[3].defaultSelected == true)
   document.write("It's true, I tell you.")
```

METHOD: Object::watch

Object.watch(property, handlerfunction)

The **watch** method is inherited by every object descended from **Object** and adds a watchpoint to a property of the method. Whenever a value is assigned to it, it calls up a function allowing you to watch any new value assigned and, if necessary, alter it.

For example, this following code watches the 'name' property of the 'city' object, and if the name 'Leningrad' is assigned to it, it is altered to the city's new name of 'St. Petersburg'. Note the code that is enclosed in the pair of curly braces (an if statement) which is associated with the **handlerfunction** argument called 'myfunction':

Code:
```
city = {name:"Chicago"}
city.watch("name", myfunction (property, oldval, newval)
                        {
                          if(newval == "Leningrad")
                            newval = "St. Petersburg"
                          return newval
                        }
           ) //end of watch method
```

NOTE:

A watchpoint for a property does not disappear if that property is deleted. It can, however, be removed by using the **unwatch** method.

OBJECT: Object

Object is the primitive Javascript object from which all other objects are derived. They therefore have all the methods defined for Object.

It is created by using the **Object** constructor as follows:

Code:
new Object()

PROPERTIES

constructor Property
This specifies a function to create an object's property and is inherited by all objects from their prototype.

Syntax: **Object.constructor**

prototype Property
This allows the addition of properties and methods to any object.

Syntax: **Object.prototype.name = value**

METHODS

eval Method
The **eval** method is deprecated as a method of **Object**, but is still used as a high level function. It evaluates a string of JavaScript in the context of an object.

Syntax: **Object.eval(string)**

toSource Method
The **toSource** method returns a literal representing the source code of an object. This can then be used to create a new object.

Syntax: **Object.toSource()**

toString Method
The **toString** method returns a string representing a specified object.

Syntax: **Object.toString()**

unwatch Method
This method removes a watchpoint set for an object and property name with the **watch** method.

Syntax: **Object.unwatch(property)**

valueOf Method
This method returns a primitive value for a specified object.

Syntax: **Object.valueOf()**

watch Method

This method adds a watchpoint to a property of the object.

Syntax: **Object.watch(property, handler)**

PROPERTY: Object::prototype

Object.prototype.name = value

Any object that can call a constructor function has a **prototype** property allowing the addition of properties and methods.

The following example first creates a 'color' property for the 'Cat' object, and then creates a specific instance of it:

Code:
```
Cat.prototype.color = null
Sheeba.color = "black"
```

METHOD: Object::toSource

Object.toSource()

The **toSource** method returns a literal representing the source code of an object. This can then be used to create a new object. Although the **toSource** method is usually called by JavaScript behind the scenes, you can call it yourself. In the case of the built-in **Object** object, it returns a string indicating that the source code is not available, while, for instances of **Object**, it returns the source. With a user-defined object, **toSource** will return the JavaScript source that defines it. The following examples illustrate these three cases:

Code:
```
Object.toSource()
```

Output:
```
function Object() { [native code] }
```

Code:
```
function Cat(breed, name, age)
{
   this.breed = breed
   this.name = name
   this.age = age
}
Cat.toSource()
```

Output:
```
function Cat(breed, name, age) { this.breed = breed; this.name = name; this.age = age; }
```

Code:
```
Sheeba = new Cat("Manx", "Felix", 7)
Sheeba.toSource()
```

Output:
```
{breed:"Manx", name:"Felix", age:7}
```

METHOD: Object::toString

Object.toString()

The **toString** method is inherited by every object descended from **Object** and returns a string representing a specified object. There are times when an object needs to be represented as a string, and the **toString** method (which comes with every object) is automatically called to do that. **ToString** returns the object type or the constructor function that created it. The following examples illustrate the use of this method and the return:

```
document.write(Sheeba)   returns   [object Object]
document.write(Sheeba.toString)   returns   function toString() { [native code] }
```

The **toString** method can, however, be overwritten in a custom object by assigning a user-defined function in its place as follows:

Code:
```
Cat.prototype.toString = myToString
```

NOTE:

Every core JavaScript object will over-ride the **toString** method to return an appropriate value, and will only call it when it needs to convert an object to a string.

METHOD: Object::valueOf

Object.valueOf()

The **valueOf** method returns a primitive value for a specified object and is inherited by all objects descended from **Object**. It is usually called automatically by JavaScript behind the scenes whenever it encounters an object where a primitive value is expected. If the object has no primitive value, then the object itself is returned as [object Object]. You can also call **valueOf** yourself to convert a built-in object into a primitive value. The following two examples illustrate uses of it:

Code:
```
(Object.valueOf()
```

Output:
```
function Object() { [native code] }
```

Code:
```
function Cat(breed, name, age)
{
   this.breed = breed
   this.name = name
}
Cat.valueOf()
```

Output:
```
function Cat(breed, name, age) { this.breed = breed this.name = name }
```

The **valueOf** method can also be overwritten in a custom object by assigning a user-defined function with no arguments in its place as follows:

Code:
```
Cat.prototype.valueOf() = myValueOf()
```

NOTE:

Every core JavaScript object will over-ride the **valueOf** method to return an appropriate value.

OBJECT: Layer

Layers provide a way to position overlapping transparent or opaque blocks of HTML content precisely on a page. When combining this functionality with JavaScript the author of a web page now has the ability to dynamically move or alter HTML elements, opening up new possibilities such as animation and zooming in/out of elements. A **Layer** object is created with every instance of the HTML <LAYER> or <ILAYER> tag in a document. These objects are then stored in the layers array of the parent document and accessed using either the name defined within the HTML tag or an integer (with '0' being the first element defined, in source order, in the specified form).

The ability to position elements above or below others requires a third positional parameter. This is called the z-index (the higher the z-index the more to the fore the layer is) and elements can be manipulated using this parameter to dynamically move them, not just horizontally and vertically, but also 'forwards' (above) and 'backwards' (below) relative to other elements on the screen.

The following example creates two layers and uses the above attribute of the layer tag to display the aboveLayer layer at the top.

Code:
```
<layer name=aboveLayer bgcolor="lightgreen" top=50 left=80 width=150 height=50> Hello
from the layer above</layer>
<layer name=belowLayer above=aboveLayer bgcolor="lightblue" top=20 left=50 width=75
height=100>Layer below</layer>
```

PROPERTIES

above Property
If a layer is topmost in the z-order, this property relates to the enclosing window object, otherwise it is a reference to the layer object above the calling layer. An example of this property is given in the code above.

Syntax: **layer.above**

background Property
This property is used to set the image used for the backdrop of a layer. The value is null if the layer has no image backdrop.

Syntax: **layer.background.src = "image"**

bgColor Property
This property defines a document's background color. The **"colorinfo"** argument is a string that can contain either the hexadecimal definition of the color or its literal description.

Syntax: **layer.bgColor = "colorinfo"**

below Property
This property represents the layer below the calling layer. The value of this property is null if the calling layer is the bottom most in the z-order.

Syntax: **layer.below**

clip.bottom Property
This property sets the bottom edge of the layer's viewable area (known as the clipping

rectangle). Anything outside of this area is not seen.

Syntax: **layer.clip.bottom**

clip.height Property
This property sets the height, in pixels, of the layer's viewable area (known as the clipping rectangle). Anything outside of this area is not seen.

Syntax: **layer.clip.height**

clip.left Property
This property sets the left edge of the layer's viewable area (known as the clipping rectangle). Anything outside of this are is not seen.

Syntax: **layer.clip.left**

clip.right Property
This property sets the right edge of the layer's viewable area (known as the clipping rectangle). Anything outside of this are is not seen.

Syntax: **layer.clip.right**

clip.top Property
This property sets the top edge of the layer's viewable area (known as the clipping rectangle). Anything outside of this are is not seen.

Syntax: **layer.clip.top**

clip.width Property
This property sets the width, in pixels, of the layer's viewable area (known as the clipping rectangle). Anything outside of this are is not seen.

Syntax: **layer.clip.width**

document Property
This property is used to access the document contained within a layer. All the methods available to the Document object can also be used to modify the contents of the layer.

Syntax: **layer.document**

left Property
This property returns the horizontal position, in pixels, of the left edge of a layer in relation to its parent layer.

Syntax: **layer.left**

name Property
This property returns a string that contains the name of a layer as defined by the ID atttribute of the <LAYER> tag.

Syntax: **layer.name**

pageX / pageYProperty
These properties return the X (horixontal) and Y (vertical) position of the specified layer in relation to the page containing it.

Syntax: **layer.pageX**
Syntax: **layer.pageY**

parentLayer Property
For a layer nested within another, this property is a reference to the parent Layer object. If not a nested layer, this property refers to the Window object that contains the layer.

Syntax: **layer.parentLayer**

siblingAbove Property
This property is a reference to the Layer object above the specified layer in z-order, amongst layers that share the same parent layer. This property's value is null if no sibling above exists.

Syntax: **layer.siblingAbove**

siblingBelow Property
This property is a reference to the Layer object below the specified layer in z-order, amongst layers that share the same parent layer. This property's value is null if no sibling above exists

Syntax: **layer.siblingBelow**

src Property
This property returns a string containing the URL of the source of the layer's content. This is the same as the SRC attribute of the <LAYER> tag.

Syntax: **layer.src**

top Property
This property is a reference to the top-most browser window that contains the specified layer. Use this property to affect changes to the layer's top-most window i.e. layerName.top.close().

Syntax: **layer.top**

visibility Property
This property determines whether or not a layer is visible.

Syntax: **layer.visibilty = "value"**

zIndex Property
This property returns the relative z-order of the specified layer in relation to any sibling layers. Any siblings with a lower zIndex are displayed below the specified layer and any with a higher zIndex are stacked above.

Syntax: **layer.zIndex**

METHODS

captureEvents Method
This method instructs the window or document to capture all events of a particular type. See the event object for a list of event types.

Syntax: **layer.captureEvent(eventType)**

handleEvent Method
This method is used to call the handler for the specified event.

Syntax: **layer.handleEvent("eventID")**

load Method

This method is used to change the contents of a layer by loading a file containing HTML code into the layer.

Syntax: **layer.load("fileName", width)**

moveAbove Method

This method is used to move the layer above the one specified with the layerName argument.

Syntax: **layer.moveAbove(layerName)**

moveBelow Method

This method is used to move the layer below the one specified with the layerName argument.

Syntax: **layer.moveBelowlayerName**

moveBy Method

This method is used to move the layer a specified number of pixels in relation to its current co-ordinates.

Syntax: **layer.moveBy(horizPixels, vertPixels)**

moveTo Method

This method moves the layer's left edge and top edge to the specified x and y co-ordinates, respectively.

Syntax: **layer.moveTo(Xposition, Yposition)**

moveToAbsolute Method

This method moves the specified layer to the pixel co-ordinates supplied in the x and y parameters, relative to the page, as opposed to the parent layer.

Syntax: **layer.moveToAbsolute(xCoord, yCoord)**

releaseEvents Method

This method is used to release any captured events of the specified type and to send them on to objects further down the event hierarchy.

Syntax: **layer.releaseEvents("eventType")**

resizeBy Method

This method is used to resize the layer by the specified horizontal and vertical number of pixels.

Syntax: **layer.resizeBy(horizPixels, vertPixels)**

resizeTo Method

This method is used to resize a layer to the dimensions supplied with the width and height (both integers, in pixels) parameters.

Syntax: **layer.resizeTo(outerWidth, outerHeight)**

routeEvent Method

This method is used to send a captured event further down the normal event hierarchy; specifically, the event is passed to the original target object unless a sub-object of the window (a document or layer) is also set to capture this type of event, in which case the event is passed to that sub-object.

Syntax: **layer.routeEvent(eventType)**

EVENT HANDLERS

onBlur Event handler

This event handler executes some specified JavaScript code on the occurrence of a **Blur** event (when an window loses focus).

Syntax: **layer.onBlur="myJavaScriptCode"**

onFocus Event handler

This event handler executes some specified JavaScript code on the occurrence of a **KeyPress** event.

Syntax: **layer.onFocus="myJavaScriptCode"**

onload Event handler

This event handler executes some specified JavaScript code on the occurrence of a **Load** event.

Syntax: **layer.onload="myJavaScriptCode"**

onMouseOut Event handler

This event handler executes some specified JavaScript code on the occurrence of a **MouseOut** event.

Syntax: **layer.onMouseOut="myJavaScriptCode"**

onMouseOver Event handler

This event handler executes some specified JavaScript code on the occurrence of a **MouseOver** event.

Syntax: **layer.onMouseOver="myJavaScriptCode"**

PROPERTY: Layer::visibility

layer.visibility = "value"

This property determines whether or not a layer is visible. The **"value"** parameter can be "show", "hide" or "inherit", the latter causing the layer to inherit the value of its parent object's visibility property.

The following example creates two layers, aboveLayer and belowLayer, and alters the visibility property of the aboveLayer layer on the occurance of a MouseOver event.

Code:
```
<layer name=aboveLayer bgcolor="lightgreen" top=50 left=80 width=150 height=50
onMouseOver=visibility="hide">

Mouse over me to reveal the layer below

</layer>

<layer name=belowLayer above=aboveLayer bgcolor="lightblue" top=20 left=20 width=150
height=50>

Hello from the layer below!

</layer>
```

METHODS

- abs
- acos
- alert
- anchor
- apply
- asin
- atan
- atan2
- back
  - History
  - Window
- blur
  - Button
  - Checkbox
  - FileUpload
  - Password
  - Radio
  - Reset
  - Select
  - Submit
  - Text
  - Textarea
  - Window
- call
- captureEvents
  - Document
  - Layer
  - Window
- ceil
- charAt
- charCodeAt
- clearInterval
- clearTimeout
- click
  - Button
  - Checkbox
  - FileUpload
  - Radio
  - Reset
  - Submit
- close
  - Document
  - Window
- compile
- concat
  - Array
  - String
- confirm
- cos
- disableExternalCapture
- enableExternalCapture
- eval
- exec
- exp
- find
- floor
- focus
  - Button
  - Checkbox
  - FileUpload
  - Password
  - Radio
  - Reset
  - Select
  - Submit
  - Text
  - Textarea
  - Window
- forward
  - History
  - Window
- fromCharCode
- getDate
- getDay
- getFullYear
- getHours
- getMilliseconds
- getMinutes
- getMonth
- getSeconds
- getSelection
- getTime
- getTimezoneOffset
- getUTCDate
- getUTCDay
- getUTCFullYear
- getUTCHours
- getUTCMilliseconds
- getUTCMinutes
- getUTCMonth
- getUTCSeconds
- go
- handleEvent
  - Button
  - Checkbox
  - Document
  - FileUpload
  - Form
  - Image
  - Layer
  - Link
  - Password
  - Radio
  - Reset
  - Select
  - Submit
  - Text
  - Textarea
  - Window
- home
- indexOf
- match
- max
- min
- moveAbove
- moveBelow
- moveBy
  - Layer
  - Window
- moveTo
  - Layer
  - Window
- moveToAbsolute
- open
  - Document
  - Window
- parse
- plugins.refresh
- pop
- pow
- preference
- print
- prompt
- push
- random
- releaseEvents
  - Document
  - Layer
  - Window
- reload
- replace
  - Location
  - String
- reset
- resizeBy
  - Layer
  - Window
- resizeTo
  - Layer
  - Window
- reverse
- round
- routeEvent
  - Document
  - Layer
  - Window scroll
- scrollBy
- scrollTo
- search
- select
  - Password
  - Text
  - Textarea
- setDate
- setFullYear
- setHours
- setInterval
- setTimeout
- setUTCDate
- setUTCFullYear
- setUTCHours
- setUTCMilliseconds
- setUTCMinutes
- setUTCMonth
- setUTCSeconds
- shift
- sin
- slice
  - Array
  - String
- sort
- splice
- split
- sqrt
- stop
- String formatting
- submit
- substr
- substring
- taintEnabled
- tan
- test
- toLocaleString
- toLowerCase
- toSource
  - Array
  - Boolean
  - Date
  - Function
  - Number
  - RegExp
  - String
- toString
  - Array
  - Boolean
  - Date
  - Function
  - Number
  - Object
  - RegExp
  - String
- toUpperCase
- toUTCString
- unshift
- unwatch
- UTC
- valueOf
  - Array
  - Boolean
  - Date
  - Function
  - Number

OBJECT: Math

The **Math** object is a top-level, built-in JavaScript object which can be accessed without using a constructor or calling a method. It also has static properties and methods for mathematical constants and functions. This means that you can refer to, say, the constant PI as **Math.PI**, and you can call the Tangent function with **Math.tan(x)**. To illustrate this, the following code calculates the length of the side of a right-angled triangle opposite the angle Theta:

Code:
```
len = Math.tan(theta) * adj
```

All constants are defined as precision real numbers in JavaScript. When using several Math constants and methods, it is often more convenient to use the **with** statement to avoid having to repeatedly type the word Math:

Code:
```
with(Math)
{
   a = 28.27
   adj = sqrt(a/PI)
   len = adj * tan(1.1071)
}
```

PROPERTIES

E Property
This property is Euler's constant and the base of natural logarithms (approximately 2.7183)

Syntax: **Math.E**

LN10 Property
This property is the natural logarithm of 10, (approximately 2.3026).

Syntax: **Math.LN10**

LN2 Property
This property is the natural logarithm of 2, which is approximately 0.6931.

Syntax: **Math.LN2**

LOG10E Property
This property is the base 10 logarithm of E (approximately 0.4343).

Syntax: **Math.LOG10E**

LOG2E Property
This property is the base 2 logarithm of E (approximately 1.4427).

Syntax: **Math.LOG2E**

PI Property
This property is the ratio of the circuference of a circle to its diameter (approximately 3.1416).

Syntax: **Math.PI**

SQRT1_2 Property
This property is the value of 1 divided by the square root of 2 and is approximately equal to 0.7071.

Syntax: **Math.SQRT1_2**

SQRT2 Property
This property is the square root of 2 (approximately 1.4142).

Syntax: **Math.SQRT2**

METHODS

abs Method
This method returns the absolute value of a number.

Syntax: **Math.abs(x)**

acos Method
This method returns the arccosine of a number as a numeric value between 0 and PI radians. Passing it a value for 'x' which is outsite the range -1 to 1 will cause the Netscape browser to return **NaN**, and the Internet Explorer browser to return an error message. Passing it -1 will return the value of PI.

Syntax: **Math.acos(x)**

asin Method
This method returns the arcsine of a number as a numeric value between
-PI/2 and PI/2 radians. Passing it a value for 'x' which is outsite the range -1 to 1 will cause the Netscape browser to return **NaN**, and the Internet Explorer browser to return an error message. Passing it 1 will return the value of PI/2.

Syntax: **Math.asin(x)**

atan Method
This method returns the arctangent of a number as a numeric value between -PI/2 and PI/2 radians.

Syntax: **Math.atan(x)**

atan2 Method
This method returns the arctangent of the quotient of its arguments.

Syntax: **Math.atan2(y, x)**

ceil Method
This method returns an integer equal to, or the next integer greater than, the number passed to it. Hence, if you passed it 3.79, it would return 4, and passing it -3.79 would return -3.

Syntax: **Math.ceil(x)**

cos Method
This method returns the cosine of a number, which will be a numeric value between -1 and 1.

Syntax: **Math.cos(x)**

exp Method
This method returns the value of E^x where E is Euler's constant and x is the argument passed to it.

Syntax: **Math.exp(x)**

floor Method
This method returns an integer equal to, or the next integer less than, the number passed to it. Hence, if you passed it 3.79, it would return 3, and passing it -3.79 would return -4.

Syntax: **Math.floor(x)**

log Method
This method returns the natural logarithm (base E) of a number. If you pass the **log** method the number 0, the Netscape browser will return -Infinity, and with an argument of a negative number **NaN**. In both these cases Internet Explorer returns an error message.

Syntax: **Math.log(x)**

max Method
This method returns the greater of the two numbers passed to it as arguments. Hence, if you passed it the numbers 9 and 11, it would return 11, whereas passing it -9 and -11 returns -9.

Syntax: **Math.max(x, y)**

min Method
This method returns the lesser of the two numbers passed to it as arguments. Hence, if you passed it the numbers 9 and 11, it would return 9, whereas passing it -9 and -11 returns -11.

Syntax: **Math.min(x, y)**

pow Method
This method returns the value of x to the power of y (x^y), where x is the base, and y is the exponent.

Syntax: **Math.pow(x, y)**

random Method
This method takes no arguments and returns a pseudo-random number between 0 and 1. The random number generator is seeded from the current time.

Syntax: **Math.random()**

round Method
This method is used to round a number to the nearest integer. If the fractional portion of the number is .5 or higher, then the number is rounded up, otherwise it is rounded down.

Syntax: **Math.round(x)**

sin Method
This method is used to return the sine of its argument, which will be a number between -1 and 1.

Syntax: **Math.sin(x)**

sqrt Method
This method returns the square root of a number. If that number is negative, then the Netscape browser returns the value of **NaN**, whereas the Internet Explorer browser returns an Error message.

Syntax: **Math.sqrt(x)**

tan Method
This method returns a number representing the tangent of an angle.

Syntax: **Math.tan(x)**

METHOD: Math::atan2

Math.atan2(y, x)

The **atan2** method returns the arctangent of the quotient of its two arguments. Compare this to the **atan** method whose single argument is the ratio of these two co-ordinates. Specifically, the angle returned is a numeric value between PI and -PI and represents the counterclockwise angle in radians (not degrees) between the positive X axis and the point (x, y).

NOTE:

The y co-ordinate is passed as the first argument and the x co-ordinate is passed as the second argument, i.e. atan2(y, x). This is purposeful and is agreement with the ECMA-262 standard.

Assuming you had a point with the (x, y) co-ordinates of (3, 6), you could calculate the angle in radians between that point and the positive X axis as follows:

Code:
```
Math.atan2(6, 3)
```

output:
```
0.4636476090008061
```

This is equivalent to calculating the arctangent of the ratio of these two co-ordinates, which is 6/3 = 2, as follows:

Code:
```
Math.atan(2)
```

output:
```
0.4636476090008061
```

METHOD: Window::alert

window.alert("message")

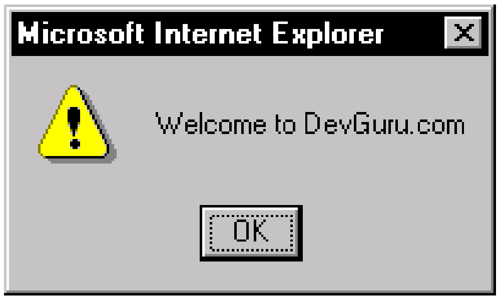
This method is used to display an alert box containing a message and an o.k. button. Use this method to convey a message that does not require a decision from the user.

Code:
```
window.alert("Welcome to DevGuru.com")
```

If you wish to have the text appear on more than one line, you use the **\n** as a line break.

```
window.alert("Welcome to\nDevGuru.com")
```

Output:


Welcome to DevGuru.com

METHOD: String::anchor

object.anchor("name")

This method is used to create an HTML anchor in a document.

The following code creates an HTML anchor called "newAnchor" and writes the contents of the "myString" **String** object to the document which can then be used as a target for a hyperlink. This has identical results to using the HTML code: <A NAME="newAnchor">Anchor</A>

Code:

```
myString = new String("Anchor")
document.write (myString.anchor("newAnchor"))
```

METHOD: Function::apply

Function.apply(thisArg[, argArray])

The **apply** method allows you to call a function and specify what the keyword **this** will refer to within the context of that function. The thisArg argument should be an object. Within the context of the function being called, **this** will refer to thisArg. The second argument to the **apply** method is an array. The elements of this array will be passed as the arguments to the function being called. The argArray parameter can be either an array literal or the deprecated **arguments** property of a function.

The **apply** method can be used to simulate object inheritance as in the following example. We first define the constructor for an object called Car which has three properties. Then the constructor for a second object called RentalCar is defined. RentalCar will inherit the properties of Car and add one additional property of its own - carNo. The RentalCar constructor uses the **apply** method to call the Car constructor, passing itself as thisArg. Therefore, inside the Car function, the keyword **this** actually refers to the RentalCar object being constructed, and not a new Car object. By this means,the RentalCar object inherits the properties from the Car object.

Code:

```
function Car(make, model, year)
{
  this.make = make;
  this.model = model;
  this.year = year;
}

function RentalCar(carNo, make, model, year)
{
  this.carNo = carNo;
  Car.apply(this, new Array(make, model, year))
}

myCar = new RentalCar(2134,"Ford","Mustang",1998)
document.write("Your car is a " + myCar.year + " " +
  myCar.make + " " + myCar.model + ".")
```

Output:

```
Your car is a 1998 Ford Mustang.
```

NOTE: The **apply** method is very similar to the **call** method and only differs in that, up until now, you could use the deprecated **arguments** array as one of its parameters.

PROPERTY: Function::arguments

[Function.]arguments

The **arguments** property consists of an array of all the arguments passed to a function. The **arguments** property is only available inside a function, but can be used to refer to any of the function's arguments by stating the appropriate element of the array. For example; **arguments[0]**; **newFunction.arguments[1]** etc. (Note that the **arguments** property can be preceeded by the function name). The **arguments** array is especially useful with functions that can be called with a variable number of arguments, or with more arguments than they were formally declared to accept.

In this next example, a function is declared to calculate the average of a variable number of numbers (which are the function's arguments). By using the **arguments** array and the **arguments.length** property, you can pass the function any number of arguments and have it return the average of them:

Code:
```
function calcAverage()
{
  var sum = 0
  for(var i=0; i<arguments.length; i++)
    sum = sum + arguments[i]
  var average = sum/arguments.length
  return average
}
document.write("Average = " + calcAverage(400, 600, 83))
```

Output:
```
Average = 361
```

The **arguments** property itself has the following three properties:

PROPERTIES

arguments.callee Property
The **arguments.callee** property can only be used within the body of a function and returns a string specifying what that function is. As the **this** keyword doesn't refer to the current function, you can use the **arguments.callee** property instead.

Syntax: **[Function.]arguments.callee**

The next example demonstrates the use of this property:

Code:
```
function testCallee(){return arguments.callee}
document.write(testCallee())
```

Output:
```
function testCallee(){return arguments.callee}
```

arguments.caller Property
The **arguments.caller** property is deprecated in JavaScript 1.3 and is no longer used, but

where it is, it specifies the name of the function that called the currently executing function.

arguments.length Property
The **arguments.length** property returns the number of arguments passed to a function, as opposed to the **function.length** property, which returns the number of arguments that a function expects to receive.

Syntax: **[Function.]arguments.length**

The distinction between the **arguments.length** and **Function.length** properties is demonstrated in this next example of a function which is designed to take as its arguments 3 numbers and then calculate the average of them. If exactly 3 arguments are passed to it, it carries out the calculation, otherwise it returns an appropriate message:

Code:
```
function calc3Average(x, y, z)
{
   if(arguments.length != calc3Average.length)
      return "Use 3 arguments!"
   else
      var average = (x + y + z)/3
      return "The average is " + average
}
```

OBJECT: Function

new Function([arg1[, arg2[, ... argN]],] functionBody)

The **Function** object permits a function to have methods and properties associated with it. To accomplish this, the function is temporarily considered to an object whenever you wish to invoke a method or read the value of a property.

Note that JavaScript treats the function, itself, as a data type that has a value. To return that value, the function must have a **return** statement.

When a **Function** object is created by using the **Function** constructor, it is evaluated each time. This is not as efficient as the alternative method of declaring a function using the **function** statement where the code is compiled.

The following example creates a **Function** object to calculate the average of two numbers, and then displays that average:

Code:
```
var twoNumAverage = new Function("x", "y", "return (x + y)/2")
document.write(twoNumAverage(3,7))
```

The **Function** object can be called just as if it were a function by specifying the variable name:

Code:
```
var average = twoNumAverage(12,17)
```

If a function changes the value of a parameter, this change is not reflected globally or in the calling function, unless that parameter is an object, in which case any changes made to any if its properties will be reflected outside of it. In the next example, an object called TaxiCo is created with a property containing the size of the taxi fleet. A **Function** object called AddCar is then created which allows a user to alter the size of the Fleet property to reflect an increase in the number of cars, and then an instance of this function adds 2 to the Fleet property with the final line of code displaying the new size of the taxi fleet:

Code:
```
taxiCo = {name:"City Cabs", phone:"321765", fleet:17}
var addCar = new Function("obj", "x", "obj.fleet = obj.fleet + x")
addCar(taxiCo, 2)
document.write("New fleet size = " + taxiCo.fleet)
```

Output:
```
New fleet size = 19
```

The function in the above example could also be created by declaring it using the **function** statement as follows...

```
function addCar(obj,x){obj.fleet = obj.fleet + x}
```

...the difference being that when you use the **Function** constructor, AddCar is a variable whose value is just a reference to the function created, whereas with the **function** statement AddCar is not a variable at all but the name of the function itself.

You can also nest a function within a function in which case the inner function can only be accessed by statements in the outer function. The inner function can use arguments and variables of the outer function, but not vice versa. The following example has an inner function that converts a monetary value from Pounds Sterling into Dollars. The outer function takes four values (the first two in Dollars and the second two in Pounds), passes each of the Pound values to the inner function to be converted to Dollars, and then adds them all together returning the sum:

Code:
```
function totalDollars(v,w,x,y)
{
  function convertPounds(a)
  {
    return a * 1.62
  }
  return v + w + convertPounds(x) + convertPounds(y)
}
document.write("Total Dollars = " + totalDollars(400, 560, 250, 460))
```

Output:
```
Total Dollars = 2110.2
```

A **Function** object can also be assigned to an event handler (which must be spelled in lowercase) as in the following example:

Code:
```
window.onmouseover = new Function("document.bgColor='lightgreen'")
```

A **Function** object can be assigned to a variable, which in turn can then be assigned to an event handler, provided it doesn't take any arguments, because event-handlers cannot handle them. In the following example a function which changes the background color to green is assigned to a variable ChangeBGColor. This in turn is assigned to an **onMouseOver** event connected to an anchor in a line of text:

Code:
```
<script language="javascript">
var changeBGColor = new Function("document.bgColor='lightgreen'")
</script>
He turned <a name="ChangeColor" onMouseOver="changeBGColor()">green </a> with envy.
```

PROPERTIES

arguments Property

The **arguments** property consists of an array of all the arguments passed to a function.

Syntax: **[Function.]arguments**

arguments.callee Property

The **arguments.callee** property can only be used within the body of a function and returns a string specifying what that function is.

Syntax: **[Function.]arguments.callee**

arguments.caller Property

The **arguments.caller** property is deprecated in JavaScript 1.3 and is no longer used, but where it is, it specifies the name of the function that called the currently executing function.

Syntax: **[Function.]arguments.caller**

arguments.length Property
The **arguments.length** property returns the number of arguments passed to a function.

Syntax: **[Function.]arguments.length**

arity Property
The **arity** property specifies the number of arguments expected by a function. It is external to the function and is in contrast to the **arguments.length** property which specifies the number of arguments actually passed to a function.

Syntax: **[Function.]arity** Compare the **length** property below.

constructor Property
The **constructor** property specifies the function that creates an object's prototype. It is a direct reference to the function itself rather than a string containing the function's name. See the **constructor** property of the **Object** object for more details and examples.

Syntax: **Function.constructor**

length Property
The **length** property specifies the number of arguments expected by a function. It is external to the function and is in contrast to the **arguments.length** property which specifies the number of arguments actually passed to a function. Compare the **arity** property above.

Syntax: **Function.length**

prototype Property
The **prototype** property is a value from which all instances of an object are constructed, and which also allows you to add other properties and methods to an object. See also the **prototype** property of the **Object** object for more details and examples.

Syntax: **Function.prototype.name = value**

METHODS

apply Method
The **apply** method allows you to apply to a function a method from another function.

Syntax: **Function.apply(thisArg[, argArray])**

call Method
The **call** method allows you to call a method from another object

Syntax: **Function.call(thisArg[, arg1[, arg2[, ...]]])**

toSource Method
The **toSource** method creates a string representing the source code of the function. This over-

rides the **Object.toSource** method

Syntax: **Function.toSource ()**

toString Method

The **toString** method (like the **valueOf** method below) returns a string which represents the source code of a function. This over-rides the **Object.toString** method.

Syntax: **Function.toString ()**

valueOf Method

The **valueOf** method (like the **ToString** method above) returns a string which represents the source code of a function. This over-rides the **Object.valueOf** method.

Syntax: **Function.valueOf ()**

STATEMENT: return

return

The **return** statement specifies the value to be returned by a function and performs the act of returning that value to where the function was called from.

The following example returns the average of three numbers entered as arguments:

Code:
```
function average(a, b, c)
{
  return (a + b + c)/3;
}
```

STATEMENT: function

function name([param] [, param] [..., param]) {statements}

The **function** statement declares a function with its specified parameters, which can include numbers, strings and objects. To return a value, a function must have a **return** statement specifying the value to return.

The word, function, is a reserved word. You cannot name a function, "function". Rather, use a word, or blend of words, that describe the purpose that the function performs.

The following code declares a function that calculates the average of three numbers and returns the result:

Code:
```
function calcaverage(a, b, c)
{
   return (a+b+c)/3;
}
document.write(calcaverage(2, 4, 6));
```

METHOD: Function::call

Function.call(thisArg[, arg1[, arg2[, ...]]])

The **call** method allows you to call a method from another object. This means you only need to write an object once and just apply it to any other objects that make use of it. The following example first creates an object called Car which has three properties. Then a second object is created called HireCar which (beside others) also has those same properties. So, instead of having to rewrite those properties, the HireCar object uses the **call** method to inherit them from the Car object. Note that that you can assign a different **this** object when calling an existing function.

Code:

```
function car(make, model, year)
{this.make = make, this.model = model, this.year = year}

function hireCar(carNo, make, model, year)
{this.carNo = carNo, car.call(this, make, model, year)}
```

NOTE

The **call** method is very similar to the **apply** method, but differs in that with **call** you cannot have the now deprecated **arguments** array as one of its parameters.

METHOD: Function::toSource

Function.toSource ()

The **toSource** method creates a string representing the source code of the function. This overrides the **Object.toSource** method. Although the **toSource** method is usually called by JavaScript behind the scenes, you can call it yourself. This can be particularly useful when debugging. In the case of the built-in **Function** object, it returns a string indicating that the source code is not available, while, for a user-defined function, **toSource** will return the JavaScript source that defines it as a string. The following examples illustrate these uses:

```
Function.toSource()   returns:
function Function() { [native code] }

function Cat(breed, name, age)
{
   this.breed = breed
   this.name = name
   this.age = age
}
Cat.toSource()   returns:
function Cat(breed, name, age) { this.breed = breed; this.name = name; this.age = age; }

Sheeba = new Cat("Manx", "Felix", 7)
Sheeba.toSource()   returns:
{breed:"Manx", name:"Felix", age:7}
```

METHOD: Function::toString

Function.toString ()

The **toString** method returns a string which represents the source code of a function. This overrides the **Object.toString** method. There are times when a function needs to be represented as a string, and the **toString** method is automatically called to do that. In this next example, the **document.write** statement requires the function Car to be represented as a string, so JavaScript automatically calls the **toString** method to produce the following output:

Code:
```
function car(make, model, year)
{this.make = make, this.model = model, this.year = year}
document.write(car)
```

Output:
```
function car(make, model, year) {this.make = make, this.model = model, this.year = year
```

With the built-in **Function** object the **toString** method would produce the following string:
function Function() { [native code] }.

METHOD: Function::valueOf

Function.valueOf ()

The **valueOf** method returns a string which represents the source code of a function. This overrides the **Object.valueOf** method. The **valueOf** method is usually called by JavaScript behind the scenes, but to demonstrate the output, the following code first creates a **Function** object called Car, and then displays the value of that function:

Code:
```
function car(make, model, year)
{this.make = make, this.model = model, this.year = year}
document.write(car.valueOf())
```

Output:
```
function car(make, model, year) {this.make = make, this.model = model, this.year = year
```

With the built-in **Function** object the **valueOf** method would produce the following string:

Output:
```
function Function() { [native code] }.
```

OBJECT: History

The **History** object is a predefined JavaScript object which is accessible through the **history** property of a **window** object. The **window.history** property is an array of URL strings which reflect the entries in the **History** object. The **History** object consists of an array of URLs, accessible through the browser's Go menu, which the client has visited within a window. It is possible to change a window's current URL without an entry being made in the **History** object by using the **location.replace** method.

PROPERTIES

current Property
The **current** property contains the complete URL of the current **History** entry.

Syntax: **history.current**

length Property
The **length** property contains the number of elements in the **History** list.

Syntax: **history.length**

next Property
The **next** property contains the complete URL of the next element in the **History** list, and is the equivalent of the URL the user would go to if they selected Forward in the Go menu.

Syntax: **history.next**

previous Property
The **previous** property contains the complete URL of the previous element in the **History** list, and is the equivalent of the URL the user would go to if they selected Back in the Go menu.

Syntax: **history.previous**

METHODS

back Method
The **back** method loads the previous URL in the **History** list, and is equivalent to the browser's Back button and to **history.go(-1)**.

Syntax: **history.back()**

forward Method
The **forward** method loads the next URL in the **History** list, and is equivalent to the browser's Forward button and to **history.go(1)**.

Syntax: **history.forward()**

go Method
The **go** method loads a specified URL from the **History** list.

Syntax: **history.go(delta)**

history.go(location)

METHOD: History::go

history.go(delta)

history.go(location)

The **go** method loads a specified URL from the **History** list. There are two ways of doing this: you can either go to a relative position backwards or forwards in the list, or you can specify all or part of the URL you wish to load. To go to a relative position forwards in the list, you use as the Delta argument a number greater than 0 for the number of places forwards and, likewise, to go to a relative position backwards in the list, you need to specify a negative number equal to the number of places backwards. If the Delta argument is 0, then the current page is reloaded. for example, the following code creates a button, which when pressed, loads a page 2 entries previous in the history list:

Code:
```
<INPUT TYPE="button" VALUE="Go" onClick="history.go(-2)">
```

In the following example, a button is created, which when pressed, loads the nearest **History** entry that contains the string "home.newco.com":

Code:
```
<INPUT TYPE="button" VALUE="Go" onClick="history.go('home.newco.com')">
```

METHOD: Window::back

window.back()

Using this method is the same as clicking the browser's Back button, i.e. it undoes the last navigation step performed from the current top-level window.

The following example creates a button on the page that acts the same as the browser's back button.

Code:

```
<input type="button" value="Go back" onClick="window.back()">
```

Output:

Back

OBJECT: Password

A **Password** object is created by using an HTML <INPUT> tag and assigning "password" to the TYPE attribute. When a user then enters a password, an asterisk (*) is displayed for every character entered, thus hiding the value of the password from the view of others. A **Password** object must be defined within an HTML <FORM> tag and the JavaScript runtime engine will then create an entry for it in the **elements** property of the appropriate **Form** object. The **Password** object can then be accessed through this **elements** array by referencing either its element number or name, if a NAME attribute was used in its creation. For example, the following HTML code creates a password field with the name "Pass" and no initial value:

Code:
```
<input type = "password" name = "pass" value = "" size = 20>
```

Using javaScript you could then, say, test the value of a user's entry in the password field as in the following example which, if the user's entry matches the value previously stored in the MyPassWord variable, executes a function called AllowEntry.

Code:
```
if(document.myForm.pass.value == myPassWord)
   allowEntry()
```

NOTE:

If a user alters a password interactively, it can only be evaluated accurately if data-tainting is enabled.

PROPERTIES

defaultValue Property
This property, tainted by default, is a string reflecting the VALUE attribute of a **Password** object. Initially this is **null** (for security reasons) regardless of any value assigned to it. You can override the initial **defaultValue** property at any time by setting it programmatically, although this won't be reflected in the display of the **Password** object.

Syntax: **object.defaultValue**

form Property
This property is a reference to the parent form to which a particular **Password** object belongs.

Syntax: **object.form**

name Property
This property, which is tainted by default, is a string reflecting the NAME attribute of a **Password** object, and can be set at any time, overriding the previous value.

NOTE:

If more than one object on any form share the same NAME attribute, an array of those objects is automatically created.

Syntax: **object.name**

type Property
This property reflects the type of any particular object on a form, and in the case of the **Password** object is always "password".

Syntax: **object.type**

value Property
This property, tainted by default, reflects the value entered into a password field by the user. It can be set programatically at any time, but if a user tries to alter it interactively, it won't be evaluated properly unless data-tainting is enabled. Whether altered or not, the value is at all times displayed as a string of asterisks.

Syntax: **object.value**

METHODS

blur method
This method is used to remove focus from the object.

Syntax: **object.blur()**

focus method
This method is used to give focus to an object. It can be used to focus on a **Password** object prior to a value being entered, either by the user in the password field, or by JavaScript code programatically.

Syntax: **object.focus()**

handleEvent method
This method calls the handler for a specified event.

Syntax: **object.handleEvent(event)**

select method
This method causes the input area of a **Password** object to be selected and the cursor to be positioned ready for user input.

Syntax: **object.select()**

NOTE:

The **Select** object also inherits the **watch** and **unwatch** methods from the **Object** object.

EVENT HANDLERS

onBlur EventHandler
This event handler causes JavaScript code to be executed whenever a blur event occurs; i.e. whenever a window, frame or form element loses focus.

Syntax: **onBlur = "myJavaScriptCode"**

onFocus EventHandler
This event handler executes JavaScript whenever a focus event occurs; i.e. whenever the user focuses on a window, frame or frameset, or inputs to a form element.

Syntax: **onFocus = "myJavaScriptCode"**

OBJECT: Radio

The **Radio** object represents a single radio button, created by an HTML <INPUT> tag of type "radio", in a series from which the user may select only one. It is for this reason that all radio buttons in a group must have the same value for the NAME attribute. The **Radio** object is a form element and as such must be included within a <FORM> tag. The JavaScript runtime engine creates a **Radio** object for each individual radio button on the form. Those which form a group, and hence share the same NAME attribute, are stored as an array of that name. This array is in turn stored as a single element of the **elements** array of the **Form** object. You can access a set of radio buttons through this **elements** array either by the index number or by name. To access an individual button you need to refer to it as an element of the array of those buttons with the same NAME attribute.

For instance, assuming a set of radio buttons with the name "drink", you could refer to it as follows:

Code:
```
document.myForm.drink
```

...or by its number in the **elements** array of the appropriate form (assume it to be 3 in this case):

Code:
```
document.myForm.elements[3]
```

Then, say, to display the value of the radio button at element # 1 of the "drink" array, you could use the following code:

Code:
```
document.write(document.myForm.drink[1].value)
```

PROPERTIES

checked Property
This property, which is by default tainted, is a Boolean value which reflects whether a particular radio button has been selected, returning **true** if it has and **false** if not. As only one radio button in a set can be selected at any one time, it follows that if the **select** property of one is **true**, then that property for the others of that set is **false**. The **checked** property can be set at any time and the change is immediately reflected in the display.

Syntax: **object.checked**

defaultChecked Property
This property, which is by default tainted, is a Boolean value initially reflecting whether a particular radio button was selected by default using the CHECKED attribute, returning **true** if it was, and **false** if not. The **defaultChecked** property can be set at any time, but the change is not displayed, nor does it affect the **defaultChecked** property of any other radio button in the set.

Syntax: **object.defaultChecked**

form Property
This property is a reference to the parent form of a set of radio buttons that share the same

NAME attribute.

Syntax: **object.form**

name Property
This property, tainted by default, refers to the NAME attribute of the set to which one particular radio button belongs. The **name** property can be set at any time but doing so places a radio button in a different group.

Syntax: **object.name**

type Property
This property specifies the type of an element in a form and reflects the TYPE attribute. In the case of a set of radio buttons, this is "radio".

Syntax: **object.type**

value Property
It is this property that is returned to the server when a radio button is selected and the form submitted. It is not displayed and so is not necessarily the same as any text that may appear alongside the radio button. The **value** property is tainted by default and reflects the VALUE attribute of the HTML code. Where no value is specified, the **value** property is the string "on".

Syntax: **document.value**

METHODS

blur Method
This method removes focus from a selection list.

Syntax: **object.blur()**

click Method
This method programatically triggers a radio buttons **onClick** event handler.

Syntax: **object.click()**

focus Method
This method moves the focus to the specified selection list allowing the user to then select from it.

Syntax: **object.focus()**

handleEvent Method
This method calls the handler for a specified event.

Syntax: **object.handleEvent(event)**

NOTE:

The **Radio** object also inherits the **watch** and **unwatch** methods from the **Object** object.

EVENT HANDLERS

onBlur EventHandler

This event handler causes JavaScript code to be executed whenever a blur event occurs; i.e. whenever a window, frame or form element loses focus.

Syntax: **onBlur = "myJavaScriptCode"**

onClick EventHandler

The **onClick** event handler executes javaScript code whenever the user clicks (i.e. when the mouse button is pressed and released) on a **Form** object.

Syntax: **onClick = "myJavaScriptCode"**

onFocus EventHandler

This event handler executes JavaScript whenever a focus event occurs; i.e. whenever the user focuses on a window, frame or frameset, or inputs to a form element.

Syntax: **onFocus = "myJavaScriptCode"**

OBJECT: Reset

A **Reset** object (a 'Reset' button) is created with every instance of an HTML <INPUT> tag (with a 'type' value set as 'RESET') on a form. Clicking this button resets the values of all form elements back to their defaults. These objects are stored in the elements array of the parent form and accessed using either the name defined within the HTML tag or an integer (with '0' being the first element defined, in source order, in the specified form).

PROPERTIES

form Property
This property returns a reference to the **Reset** object's parent **form**.

Syntax: **object.form**

name Property
This property sets or returns the value of the **Reset** object's **name** attribute.

Syntax: **object.name**

type Property
Every element on a form has an associated **type** property. In the case of a **Reset** object, the value of this property is always "reset".

Syntax: **object.type**

value Property
This property sets or returns the **Submit** object's **value** attribute. This is the text that is actually displayed on the button face. If this is not defined within the HTML tag the string 'Reset' is used by default.

Syntax: **object.value**

METHODS

blur Method
This method removes the **focus** from the specified reset button.

Syntax: **object.blur( )**

click Method
This method simulates a mouse-click on the reset button.

Syntax: **object.click( )**

focus Method
This method gives focus to the specified reset button.

Syntax: **object.focus( )**

handleEvent Method
This method calls the handler for the specified event.

Syntax: **object.handleEvent(event)**

EVENT HANDLERS

onBlur Event handler

This event handler executes some specified JavaScript code on the occurrence of a **blur** event (when the **Reset** object loses focus).

Syntax: **object.onBlur="myJavaScriptCode"**

onClick Event handler

This event handler executes some specified JavaScript code on the occurrence of a **click** event (when the reset button is clicked).

Syntax: **object.onClick="myJavaScriptCode"**

onFocus Event handler

This event handler executes some specified JavaScript code on the occurrence of a **focus** event (when the reset button receives focus).

Syntax: **object.onFocus="myJavaScriptCode"**

OBJECT: Submit

A **Submit** object (a 'Submit' button) is created with every instance of an HTML <INPUT> tag (with a 'type' value set as 'SUBMIT') on a form. These objects are then stored in the elements array of the parent form and accessed using either the name defined within the HTML tag or an integer (with '0' being the first element defined, in source order, in the specified form).

PROPERTIES

form Property
This property returns a reference to the **Submit** object's parent **form**.

Syntax: **object.form**

name Property
This property sets or returns the value of the **Submit** object's **name** attribute.

Syntax: **object.name**

type Property
Every element on a form has an associated **type** property. In the case of a **Submit** object, the value of this property is always "submit".

Syntax: **object.type**

value Property
This property sets or returns the **Submit** object's **value** attribute. This is the text that is actually displayed on the button face. If this is not defined within the HTML tag the string 'Submit Query' is used by default.

Syntax: **object.value**

METHODS

blur Method
This method removes the **focus** from the specified submit button.

Syntax: **object.blur( )**

click Method
This method simulates a mouse-click on the submit button.

Syntax: **object.click( )**

focus Method
This method gives focus to the specified submit button.

Syntax: **object.focus( )**

handleEvent Method
This method calls the handler for the specified event.

Syntax: **object.handleEvent(event)**

EVENT HANDLERS

onBlur Event handler

This event handler executes some specified JavaScript code on the occurrence of a **blur** event (when the **Submit** object loses focus).

Syntax: **object.onBlur="myJavaScriptCode"**

onClick Event handler

This event handler executes some specified JavaScript code on the occurrence of a **click** event (when the submit button is clicked).

Syntax: **object.onClick="myJavaScriptCode"**

onFocus Event handler

This event handler executes some specified JavaScript code on the occurrence of a **focus** event (when the submit button receives focus).

Syntax: **object.onFocus="myJavaScriptCode"**

OBJECT: Text

A **Text** object provides a field on a form in which the user can enter data and is created with every instance of an HTML <INPUT> tag with the type attribute set to "text". These objects are then stored in the elements array of the parent form and accessed using either the name defined within the HTML tag or an integer (with '0' being the first element defined, in source order, in the specified form).

PROPERTIES

defaultValue Property
This property sets or returns a string indicating the initial value of the **Text** object. The value of this property initially reflects the value between the start and end <TEXT> tags. Use of the defaultValue property, which can be done at any time, will override the original value.

Syntax: **object.defaultValue[= "newdefaultvalue"]**

form Property
This property returns a reference to the parent **Form** of the **Text** object.

Syntax: **object.form**

name Property
This property sets or returns the value of the **Text** object's **name** attribute.

Syntax: **object.name**

type Property
Every element on a form has an associated **type** property. In the case of a **Text** object, the value of this property is always "text".

Syntax: **object.type**

value Property
This property sets or returns the **Text** object's **value** attribute. This is the text that is actually displayed in the text field and can be set at any time with any changes being immediately displayed.

Syntax: **object.value**

METHODS

blur Method
This method removes the **focus** from the specified **Text** object.

Syntax: **object.blur( )**

focus Method
This method gives focus to the specified **Text** object.

Syntax: **object.focus( )**

handleEvent Method
This method calls the handler for the specified event.

Syntax: **object.handleEvent(event)**

select Method
This method is used to select and highlight the entire text that is currently in the input field. When the user starts typing, they will replace whatever is currently there. Used in conjunction with the focus method, this makes it easy to prompt the user for input and places the cursor in the correct place.

Syntax: **object.select( )**

EVENT HANDLERS

onBlur Event handler

This event handler executes some specified JavaScript code on the occurrence of a **blur** event (when the **Text** object loses focus).

Syntax: **object.onBlur="myJavaScriptCode"**

onChange Event handler

This event handler executes some specified JavaScript code on the occurrence of a **change** event (when the **Text** object loses focus and its value has altered).

Syntax: **object.onChange="myJavaScriptCode"**

onFocus Event handler

This event handler executes some specified JavaScript code on the occurrence of a **focus** event (when the **Text** object receives focus).

Syntax: **object.onFocus="myJavaScriptCode"**

onSelect Event handler

This event handler executes some specified JavaScript code on the occurrence of a **select** event (when some text in the text field is selected).

Syntax: **object.onSelect="myJavaScriptCode"**

EVENT HANDLER: onSelect

onSelect = myJavaScriptCode

Event handler for **Text**,**Textarea**

The **onSelect** event handler is used to execute specified JavaScript code whenever the user selects some of the text within a text or textarea field. The **onSelect** event handler uses the following properties of the **Event** Object:

type - indicates the type of event.
target - indicates the target object to which the event was sent.

In the following example, selecting some of the text in the **Text** object causes the 'selectEvent' function to execute:

Code:
```
<INPUT TYPE="text" onSelect="selectEvent()">
```

OBJECT: Textarea

A **Textarea** object provides a multi-line field in which the user can enter data and is created with every instance of an HTML <TEXTAREA> tag on a form. These objects are then stored in the elements array of the parent form and accessed using either the name defined within the HTML tag or an integer (with '0' being the first element defined, in source order, in the specified form).

PROPERTIES

defaultValue Property
This property sets or returns a string indicating the initial value of the **Textarea** object. The value of this property initially reflects the value between the start and end <TEXTAREA> tags. Use of the defaultValue property, which can be done at any time, will override the original value.

Syntax: **object.defaultValue[= "newdefaultvalue"]**

form Property
This property returns a reference to the parent **Form** of the **Textarea** object.

Syntax: **object.form**

name Property
This property sets or returns the value of the **Textarea** object's **name** attribute.

Syntax: **object.name**

type Property
Every element on a form has an associated **type** property. In the case of a **Textarea** object, the value of this property is always "textarea".

Syntax: **object.type**

value Property
This property sets or returns the **Textarea** object's **value** attribute. This is the text that is actually displayed in the Textarea and can be set at any time with any changes being immediately displayed.

Syntax: **object.value**

METHODS

blur Method
This method removes the **focus** from the specified **Textarea** object.

Syntax: **object.blur( )**

focus Method
This method gives focus to the specified **Textarea** object.

Syntax: **object.focus( )**

handleEvent Method
This method calls the handler for the specified event.

Syntax: **object.handleEvent(event)**

select Method
This method is used to select and highlight all or a portion of a text in a Textarea element. Used in conjunction with the focus method, this makes it easy to prompt the user for input and places the cursor in the correct place.

Syntax: **object.select( )**

EVENT HANDLERS

onBlur Event handler
This event handler executes some specified JavaScript code on the occurrence of a **blur** event (when the **Textarea** object loses focus).

Syntax: **object.onBlur="myJavaScriptCode"**

onChange Event handler
This event handler executes some specified JavaScript code on the occurrence of a **change** event (when the **Textarea** object loses focus and its value has altered).

Syntax: **object.onChange="myJavaScriptCode"**

onFocus Event handler
This event handler executes some specified JavaScript code on the occurrence of a **focus** event (when the **Textarea** object receives focus).

Syntax: **object.onFocus="myJavaScriptCode"**

onKeyDown Event handler
This event handler executes some specified JavaScript code on the occurrence of a **KeyDown** event (when a key is depressed).

Syntax: **object.onKeyDown="myJavaScriptCode"**

onKeyPress Event handler
This event handler executes some specified JavaScript code on the occurrence of a **KeyPress** event (when a key is depressed and held down).

Syntax: **object.onKeyPress="myJavaScriptCode"**

onKeyUp Event handler
This event handler executes some specified JavaScript code on the occurrence of a **KeyUp** event (when a key is released).

Syntax: **object.onKeyUp="myJavaScriptCode"**

onSelect Event handler
This event handler executes some specified JavaScript code on the occurrence of a **select** event (when some text in the **Textarea** is selected).

Syntax: **object.onSelect="myJavaScriptCode"**

PROPERTY: Textarea::form

textarea.form

This property returns a reference to the **Textarea** object's parent **Form**.

The following code dispays the name of the **Textarea** object's parent **Form** when it is clicked and assumes, for the purposes of this example, that the **Form** is called "myForm".

Code:

```
<TEXTAREA NAME="txtArea" VALUE="This is a Textarea object"></TEXTAREA>

<script language="javascript">
document.write ("The parent form of this Textarea is " + document.myForm.txtArea.form.name)
</script>
```

EVENT HANDLER: onKeyDown

onKeyDown = myJavaScriptCode

Event handler for **Document**, **Image**, **Link**, **TextArea**.

The **onKeyDown** event handler executes the specified JavaScript code or function on the occurance of a KeyDown event. A KeyDown event occurs when the user depresses a key.

The **onKeyDown** event handler uses the following **Event** object properties.

type - this property indicates the type of event.
target - this property indicates the object to which the event was originally sent.
layerX - the cursor location when the KeyDown event occurs.
layerY - the cursor location when the KeyDown event occurs.
pageX - the cursor location when the KeyDown event occurs.
pageY - the cursor location when the KeyDown event occurs.
screenX - the cursor location when the KeyDown event occurs.
screenY - the cursor location when the KeyDown event occurs.
which - this represents the key pressed as its ASCII value.
modifiers - lists the modifier keys (shift, alt, ctrl, etc.) held down when the KeyDown event occurs.

The following example shows the use of the **onKeyDown** event handler to display a message in the text box.

Code:
```
<body>
<form action="" method="POST" id="myForm">
<input type="text" name="myText" onKeyDown="changeVal()">

<script type="text/javascript" language="JavaScript">
s1 = new String(myForm.myText.value)

function changeVal() {
     s1 = "You pressed a key"
    myForm.myText.value = s1.toUpperCase()
}

</script>
</form>
</body>
```

OBJECT: Image

new Image([width,] [height])

The **Image** object is an image on an HTML form, created by using the HTML 'IMG' tag. Any images created in a document are then stored in an array in the **document.images** property, and it is from here that they are accessed. If you have specified a name for an image created using the HTML 'IMG' tag, you can use that name when you index the image in the **images** array. You can also use the **Image** constructor and the **new** operator to create an image object which can then be displayed within an existing cell. The main use of this is to load an image from the network so that it is in cache before it needs to be displayed.

For example the following code creates an **Image** object called MyImage:

Code:
```
myImage = new Image()
myImage.src = "C:/Personal/Mountain.gif"
```

...you could then have this image replace an existing image River on, say, the click of a button:

Code:
```
onClick="javascript:void(document.River.src = myImage.src)"
```

When one **Image** object replaces another, as in the above example, it cannot change the **width** and **height** properties of it (these are read-only for this object) and the browser displays the image with the dimensions set in the IMG tag. JavaScript can also be used to create animation by repeatedly changing the value of the **src** property. This isn't as fast as GIF animation because Javascript has to load each indivual frame as a separate file, whereas with GIF animation all the frames are contained in one file.

PROPERTIES

border Property
The **border** property is read-only, and is a string stating the width of the border of an image in pixels. For an image created using the **Image** constructor, this is 0.

Syntax: **Image.border**

complete Property
The **complete** property is read-only and returns a Boolean value indicating whether or not the browser has completed loading the image.

Syntax: **Image.complete**

height Property
The **height** property is read-only, and is a string stating the HEIGHT attribute of an IMG tag in pixels. Where an image has been created using the **Image** constructor, the height is of the image itself, not the HEIGHT value of the display.

Syntax: **Image.height**

hspace Property
The **hspace** property is read-only and specifies the HSPACE value of the IMG tag, which is the

number of pixels space between the left and right margins of an image and surrounding text. For an image created using the **Image** constructor, the value of this property is null.

Syntax: **Image.hspace**

lowsrc Property
The **lowsrc** property specifies the URL of a low-resolution version of an image relating to the LOWSRC attribute of an IMG tag. A browser first loads a low-resolution version before replacing it with the high-resolution version of the **src** property.

Syntax: **Image.lowsrc**

name Property
The **name** property is read-only and reflects the NAME attribute of an IMG tag. If the **Image** object has been created by using the **Image** constructor, the value of this property is null.

Syntax: **Image.name**

src Property
The **src** property is a string representing the URL of an image and reflects the SRC attribute of an IMG tag. The **src** property can be altered at any time, but when you do so the new image (if not the same size) is scaled to fit the height and width attributes of the IMG tag. Also, the loading of any other image into that cell is aborted, so the **Lowsrc** property should be altered before setting the **src** property.

Syntax: **Image.src**

vspace Property
The **vspace** property is read-only and specifies the VSPACE value of the IMG tag, which is the number of pixels space between the top and bottom margins of an image and surrounding text. For an image created using the **Image** constructor, the value of this property is null.

Syntax: **Image.vspace**

width Property
The **width** property is read-only, and is a string stating the WIDTH attribute of an IMG tag in pixels. Where an image has been created using the **Image** constructor, the width is of the image itself, not the WIDTH value of the display.

Syntax: **Image.width**

METHODS

handleEvent Method
The **handleEvent** method is used to evoke the handler for a specified event.

Syntax: **Image.handleEvent(event)**

EVENT HANDLERS

All the event handlers that are available with the **Image** object also have an equivalent property (spelled entirely in lower case letters) which can be used to set an image's event-handlers when created using the **Image** constructor. Assume you have a function called MyFunction which you want to set to the **onload** event-handler for an image called Ocean, you could accomplish this with the following statement:

Code:
Ocean.onload = myFunction

The same applies for all the following event-handlers.

onAbort EventHandler

The **onAbort** event handler is used to execute certain JavaScript code whenever an abort event occurs, such as when the user stops the loading of an image by clicking a link or a Stop button.

Syntax: **onAbort = "myJavaScriptCode"**

onError EventHandler

The **onError** event handler executes certain Javascript code whenever a Javascript syntax or runtime error occurs during the loading of a document or image.

Syntax: **onError = "myJavaScriptCode"**

onKeyDown EventHandler

The **onKeyDown** event handler is used to execute certain JavaScript code whenever the user depresses a key.

Syntax: **onKeyDown = "myJavaScriptCode"**

onKeyPress EventHandler

The **onKeyPress** event handler executes JavaScript code whenever the user presses or holds down a key

Syntax: **onKeyPress = "myJavaScriptCode"**

onKeyUp EventHandler

The **onKeyUp** event handler executes JavaScript code whenever the user releases a depressed key.

Syntax: **onKeyUp = "myJavaScriptCode"**

onload EventHandler

The **onload** event handler is used to execute JavaScript code whenever the browser has finished loading a window or all of the frames within a FRAMESET tag.

Syntax: **onload ="myJavaScriptCode"**

NOTE:

The event handlers **onClick**, **onMouseOut** and **onMouseOver** can also be used with the Internet Explorer browser, but not with Netscape. You can, however, use these event handlers in conjunction with the **Image** object with Netscape, if you define an **Area** object for the image, or if the IMG tag is placed within a **Link** object.

EVENT HANDLER: onAbort

onAbort = myJavaScriptCode

Event handler for Image

The **onAbort** event handler executes the specified JavaScript code or function on the occurance of an abort event. This is when a user cancels the loading of an image by either clicking stop in the browser or clicking another link before the image has loaded.

The **onAbort** event handler uses the following Event object properties.

type - this property indicates the type of event.
target - this property indicates the object to which the event was originally sent.

The following example shows the use of the **onAbort** event handler to alert the user that the 'myPic' image was not loaded.

Code:
```
<IMG NAME = "myPic" SRC = "images/myPic.gif" onAbort = "alert('Loading of image
aborted!')">
```

OBJECT: Event

An **Event** object is created automatically by JavaScript on the occurance of an event. It has various properties that provide information about the event such as event type, the position of the cursor at the time the event occured, etc. Not all of the properties relate to every type of event; the one's that do are documented in the individual event handler pages.

The following example creates a button that, when clicked, displays an alert box showing the event type (in this case a 'click' event).

Code:
```
<input type="button" value="Event type" onClick='alert("The event type is " + event.type)'>
```

PROPERTIES

data Property
This property relates to the DragDrop event and its use returns an array that contains the URLs of any dropped objects, as strings.

Syntax: **event.data**

height Property
This property relates to the height of the window or frame that contains the object that initiated the event.

Syntax: **event.height**

layerX / layerY Property
These properties returns a number that represents the horizontal/vertical position, in pixels, of the cursor relative to the layer that initiated the event, or, when passed with a resize event, it represents the object width/height. These properties are synonyms for, and interchangable with, the **x** and **y** event object properties.

Syntax: **event.layerX**
Syntax: **event.layerY**

modifiers Property
This property returns a string containing details of any modifier keys that were held down during a key or mouse event. The values of the modifier keys are as follows: ALT_MASK, CONTROL_MASK, SHIFT_MASK and META_MASK.

Syntax: **event.modifiers**

pageX / pageY Property
These properties return the horizontal/vertical position of the cursor relative to the page, in pixels, at the time the event occured.

Syntax: **event.pageX**
Syntax: **event.pageY**

screenX / screenY Property
These properties return the horizontal/vertical position of the cursor relative to the screen, in pixels, at the time the event occured.

Syntax: **event.screenX**
Syntax: **event.screenY**

target Property
This property returns a reference to the object that the event was originally sent to.

Syntax: **event.target**

type Property
This property returns a string that represents the type of the event (click, key down, etc.).

Syntax: **event.type**

which Property
This property returns a number that represents either which mouse button (1 being the left button, 2 the middle and 3 the right) was pressed or which key was pressed (its ASCII value) at the time the event occuered.

Syntax: **event.which**

width Property
This property relates to the height of the window or frame that contains the object that initiated the event.

Syntax: **event.width**

x / y Property
These properties returns a number that represents the horizontal/vertical position, in pixels, of the cursor relative to the layer that initiated the event, or, when passed with a resize event, it represents the object width/height. These properties are synonyms for, and interchangable with, the **layerX** and **layerY** event object properties.

Syntax: **event.x**
Syntax: **event.y**

METHODS

The Event object inherits the watch and unwatch methods of the Object object.

EVENT HANDLER: onError

onError = myJavaScriptCode

Event handler for **Image**, **Window**.

The **onError** event handler executes the specified JavaScript code or function on the occurance of an error event. This is when an image or document causes an error during loading. The distinction must be made between a browser error, when the user types in a non-existant URL, for example, and a JavaScript runtime or syntax error. This event handler will only be triggered by a JavaScript error, not a browser error.

As well as the **onError** handler triggering a JavaScript function, it can also be set to **onError="null"** which suppresses the standard JavaScript error dialog boxes. To suppress JavaScript error dialogs when calling a function using **onError**, the function must return true (example 2 below demonstrates this).

There are two things to bear in mind when using **window.onerror**. Firstly, this only applies to the window containing window.onerror, not any others, and secondly, **window.onerror** must be spelt all lower-case and contained within **<script>** tags; it cannot be defined in HTML (this obviously doesn't apply when using **onError** with an image tag, as in example 1 below).

The **onFocus** event handler uses the following **Event** object properties.

type - this property indicates the type of event.
target - this property indicates the object to which the event was originally sent.

The first example suppresses the normal JavaScript error dialogs if a problem arises when trying to load the specified image, while example 2 does the same, but applied to a window, by using return true in the called function, and displays a customized message instead.

Code:
```
<IMG NAME="imgFaulty" SRC="dodgy.jpg onError="null">
```

Code:
```
<script type="text/javascript" language="JavaScript">

s1 = new String(myForm.myText.value)

window.onerror=myErrorHandler

function myErrorHandler() {
alert('A customized error message')
return true
}

</script>

<body onload=nonexistantFunc()>
```

OBJECT: Window

As the top level object in the JavaScript client hierarchy, every browser window and frame has a corresponding **Window** object, created automatically with every instance of a <BODY> or <FRAMESET> tag.

PROPERTIES

closed Property

This property is used to return a Boolean value that determines if a window has been closed. If it has, the value returned is true.

Syntax: **window.closed**

defaultStatus Property

This property is used to define the default message displayed in a window's status bar.

Syntax: **window.defaultStatus(= "message")**

document Property

This property's value is the document object contained within the window. See **Document** object.

Syntax: **window.document**

frames Property

This property is an array containing references to all the named child frames in the current window.

Syntax: **window.frames (= "frameID")**

history Property

This property's value is the window's History object, containing details of the URL's visited from within that window. See **History** object.

Syntax: **window.history**

innerHeight / innerWidth Properties

These properties determine the inner dimensions of a window's content area.

Syntax: **window.innerHeight = pixelDimensions**
window.innerWidth = pixelDimensions

length Property

This property returns the number of child frames contained within a window, and gives identical results as using the length property of the **frames** array.

Syntax: **window.length**

location Property

This property contains details of the current URL of the window and its value is always the Location object for that window.

Syntax: **window.location**

locationbar Property
This property relates to the area of a browser's window that contains the details of the URL or bookmark (this is where you physically enter URL details). The locationbar property has its own property, visible, that defaults to true (visible) and can be set to false (hidden).

Syntax: **window.locationbar[.visible = false]**

menubar Property
This property relates to the area of a browser's window that contains the various pull-down menus (File, Edit, View, etc.). The menubar property has its own property, visible, that defaults to true (visible) and can be set to false (hidden).

Syntax: **window.menubar[.visible = false]**

name Property
This property is used to return or set a window's name.

Syntax: **window.name**

opener Property
When opening a window using window.open, use this property from the destination window to return details of the source window. This has many uses, for example, window.opener.close() will close the source window.

Syntax: **window.opener**

outerheight / outerwidth Property
These properties determine the dimensions, in pixels, of the outside boundary, including all interface elements, of a window.

Syntax: **window.outerheight**
Syntax: **window.outerwidth**

pageXOffset / pageYOffset Property
These properties return the X and Y position of the current page in relation to the upper left corner of a window's display area.

Syntax: **window.pageXOffset**
Syntax: **window.pageYOffset**

parent Property
This property is a reference to the window or frame that contains the calling child frame.

Syntax: **window.parent**

personalbar Property
This property relates to the browser's personal bar (or directories bar). The personalbar property has its own property, visible, that defaults to true (visible) and can be set to false (hidden).

Syntax: **window.personalbar[.visible = false]**

scrollbars Property
This property relates to the browser's scrollbars (vertical and horizontal). The scrollbars property has its own property, visible, that defaults to true (visible) and can be set to false (hidden).

Syntax: **window.scrollbars[.visible = false]**

self Property
This property is a reference (or synonym) for the current active window or frame.
Syntax: **self.property or method**

status Property
This property, which can be set at any time, is used to define the transient message displayed in a window's status bar such as the text displayed when you **onMouseOver** a link or anchor.

Syntax: **window.status(= "message")**

statusbar Property
This property relates to the browser's status bar. The statusbar property has its own property, visible, that defaults to true (visible) and can be set to false (hidden).

Syntax: **window.statusbar[.visible = false]**

toolbar Property
This property sets or returns a Boolean value that defines whether the browser's tool bar is visible or not. The default is true (visible). False means hidden. It can only be set before the window is opened and you must have UniversalBrowserWrite privilege.

Syntax: **window.toolbar[.visible = false]**

top Property
This property is a reference (or synonym) for the topmost browser window.

Syntax: **top.property or method**

window Property
This property is a reference (or synonym) for the current window or frame.

Syntax: **window.property or method**

METHODS

alert Method
This method displays an alert box containing a message and an o.k. button.

Syntax: **window.alert("message")**

back Method
Using this method is the same as clicking the browser's Back button, i.e. it undoes the last navigation step performed from the current top-level window.

Syntax: **window.back( )**

blur Method
This method is used to remove focus from the current window.

Syntax: **window.blur( )**

captureEvents Method
This method instructs the window to capture all events of a particular type. See the event object for a list of event types.

Syntax: **window.captureEvent(eventType)**

clearInterval Method
This method is used to cancel a timeout previously set with the setInterval method.

Syntax: **window.clearInterval(intervalID)**

clearTimeout Method
This method is used to cancel a timeout previously set with the setTimeout method.

Syntax: **window.clearTimeout(timeoutID)**

close Method
This method is used to close a specified window. If no window reference is supplied, the close() method will close the current active window. Note that this method will only close windows created using the open() method; if you attempt to close a window not created using open(), the user will be prompted to confirm this action with a dialog box before closing. The single exception to this is if the current active window has only one document in its session history. In this case the closing of the window will not require confirmation.

Syntax: **window.close( )**

confirm Method
This method brings up a dialog box that prompts the user to select either 'o.k.' or 'cancel', the first returning true and the latter, false.

Syntax: **window.confirm("message")**

disableExternalCapture Method
This method disables the capturing of events previously enabled using the enableExternalCapture method below.

Syntax: **window.disableExternalCapture( )**

enableExternalCapture Method
This method allows a window that contains frames to capture events in documents loaded from different servers.

Syntax: **window.enableExternalCapture( )**

find Method
This method allows the searching of the contents of a window for a specified string. The **caseSensitive** and **backward** arguments are Booleans and to use either of these you must also specify the other. If a search string is not supplied, JavaScript will display a Find dialog box which prompts the user for a string to search for, and also provides the facility to set the other two (**caseSensitive** and **backward**) arguments.

Syntax: **window.find([string[, caseSensitive, backward]])**

focus Method
This method is used to give focus to the specified window. This is useful for bringing windows to the top of any others on the screen.

Syntax: **window.focus( )**

forward Method

Using this method is the same as clicking the browser's Forward button, i.e. it goes to the next URL in the history list of the current top-level window.

Syntax: **window.forward( )**

handleEvent Method

This method is used to call the handler for the specified event.

Syntax: **window.handleEvent("eventID")**

home Method

Using this method has the same effect as pressing the Home button in the browser, i.e. the browser goes to the URL set by the user as their home page.

Syntax: **window.home( )**

moveBy Method

This method is used to move the window a specified number of pixels in relation to its current co-ordinates.

Syntax: **window.moveBy(horizPixels, vertPixels)**

moveTo Method

This method moves the window's left edge and top edge to the specified x and y co-ordinates, respectively.

Syntax: **window.moveTo(Xposition, Yposition)**

open Method

This method is used to open a new browser window.

Syntax: **window.open(URL, name [, features])**

print Method

This method is used to print the contents of the specified window.

Syntax: **window.print( )**

prompt Method

This method displays a dialog box prompting the user for some input.

Syntax: **window.prompt(message[, defaultInput])**

releaseEvents Method

This method is used to release any captured events of the specified type and to send them on to objects further down the event hierarchy

Syntax: **window.releaseEvents("eventType")**

resizeBy Method

This method is used to resize the window. It moves the bottom right corner of the window by the specified horizontal and vertical number of pixels while leaving the top left corner anchored to its original co-ordinates.

Syntax: **window.resizeBy(horizPixels, vertPixels)**

resizeTo Method

This method is used to resize a window to the dimensions supplied with the **outerWidth** and **outerHeight** (both integers, in pixels) parameters.

Syntax: **window.resizeTo(outerWidth, outerHeight)**

routeEvent Method
This method is used to send a captured event further down the normal event hierarchy; specifically, the event is passed to the original target object unless a sub-object of the window (a document or layer) is also set to capture this type of event, in which case the event is passed to that sub-object.

Syntax: **window.routeEvent(eventType)**

scroll Method
This method is used to scroll the window to the supplied co-ordinates. This method is now deprecated; use the scrollTo method detailed below instead.

Syntax: **window.scroll(coordsPixels)**

scrollBy Method
This method is used to scroll the window's content area by the specified number of pixels. This is only useful when there are areas of the document that cannot be seen within the window's current viewing area, and the visible property of the window's scrollbar must be set to true for this method to work.

Syntax: **window.scrollBy(horizPixels, vertPixels)**

scrollTo Method
This method scrolls the contents of a window, the specified co-ordinate becoming the top left corner of the viewable area.

Syntax: **window.scrollTo(xPosition, yPosition)**

setInterval Method
This method is used to call a function or evaluate an expression at specified intervals, in milliseconds.

Syntax: **window.setInterval(expression/function, milliseconds)**

setTimeout Method
This method is used to call a function or evaluate an expression after a specified number of milliseconds.

Syntax: **window.setTimeout(expression/function, milliseconds)**

stop Method
This method is used to cancel the current download. This is the same as clicking the browser's Stop button.

Syntax: **window.stop( )**

EVENT HANDLERS

onBlur Event handler
This event handler executes some specified JavaScript code on the occurrence of a **Blur** event (when an window loses focus).

Syntax: **window.onBlur="myJavaScriptCode"**

onDragDrop Event handler
This event handler executes some specified JavaScript code on the occurrence of a **DragDrop** event.

Syntax: **window.onDragDrop="myJavaScriptCode"**

onError Event handler
This event handler executes some specified JavaScript code on the occurrence of an **Error** event.

Syntax: **window.onError="myJavaScriptCode"**

onFocus Event handler
This event handler executes some specified JavaScript code on the occurrence of a **Focus** event.

Syntax: **window.onFocus="myJavaScriptCode"**

onload Event handler
This event handler executes some specified JavaScript code on the occurrence of a **Load** event.

Syntax: **window.onload="myJavaScriptCode"**

onMove Event handler
This event handler executes some specified JavaScript code on the occurrence of a **Move** event.

Syntax: **window.onMove="myJavaScriptCode"**

onResize Event handler
This event handler executes some specified JavaScript code on the occurrence of a **Resize** event.

Syntax: **window.onResize="myJavaScriptCode"**
onUnload Event handler
This event handler executes some specified JavaScript code on the occurrence of an **Unload** event.

Syntax: **window.onUnload="myJavaScriptCode"**

PROPERTY: Window::closed

window.closed

This property is used to return a Boolean value that determines if a window has been closed. If it has, the value returned is true.

The following code opens a new window and then immediately closes it. The **onClick** event of the button then calls a function which uses the **window.closed** property to display the status (open or closed) of the window.

Code:

```
<INPUT TYPE="Button" NAME="winCheck" VALUE="Has window been closed?"
onClick=checkIfClosed()>

newWindow=window.open('','','toolbar=no,scrollbars=no,width=300,height=150')
newWindow.document.write("This is 'newWindow'")
newWindow.close()

function ifClosed() {
document.write("The window 'newWindow' has been closed")
}

function ifNotClosed() {
document.write("The window 'newWindow' has not been closed")
}

function checkIfClosed() {
if (newWindow.closed)
   ifClosed()
else
   ifNotClosed()
}
```

PROPERTY: Window::defaultStatus

window.defaultStatus(= "message")

This property, which can be set at any time, is used to define the default message displayed in a window's status bar, with priority given to any other status messages such as the text displayed when you **onMouseOver** a link or anchor.

Code:

```
window.defaultStatus = "This is the default status bar message."
```

PROPERTY: Window::frames

window.frames("frameID")

This property is an array containing references to all the named child frames in the current window. These references are stored in the array in the order in which they are defined in the source code. The **"frameID"** argument is used to access items in the array and this can either be a string containing the child frame name as defined with the <FRAME> tag in the HTML source, or an integer (with '0' being the first item in the array).

The first two examples below return the same results; the first uses the defined names of the frames and the second uses their reference number within the array.

The frames array also has a length property which determines how many child frames are contained within a window. This is identical to using the length property of the window object. The third example shows the syntax for this.

Code:
```
window.frames["framename1"]
window.frames["framename2"]
window.frames["framename3"]

window.frames[0]
window.frames[1]
window.frames[2]

window.frames.length
```

PROPERTY: Window::status

window.status= ("message")

This property, which can be set at any time, is used to define the transient message displayed in a window's status bar such as the text displayed when you **onMouseOver** a link or anchor. When using the status property with the onMouseOver event handler, you must use the 'return true' syntax as detailed in the example below.

Code:
```
<A HREF="http://www.devguru.com" onMouseOver="self.status='Visit DevGuru.com'; return
true">DevGuru.com</A>
```

METHOD: Window::setInterval

window.setInterval(expression/function, milliseconds)

This method is used to call a function or evaluate an expression at specified intervals, in milliseconds. This will continue until the **clearInterval** method is called or the window is closed. If an expression is to be evaluated, it must be quoted to prevent it being evaluated immediately

The following example uses the **setInterval** method to call the clock() function which updates the time in a text box.

Code:
```
<form name="myForm" action="" method="POST">
<input name="myClock" type="Text">
<script language=javascript>

self.setInterval('clock()', 50)

function clock() {
   time=new Date()
   document.myForm.myClock.value=time
}

</script>
</form>
```

METHOD: Window::setTimeout

window.setTimeout(expression/function, milliseconds)

This method is used to call a function or evaluate an expression after a specified number of milliseconds. If an expression is to be evaluated, it must be quoted to prevent it being evaluated immediately. Note that the use of this method does not halt the execution of any remaining scripts until the timeout has passed, it just schedules the expression or function for the specified time.

The following example opens a new window and uses the **setTimeout** method to call the winClose() function which closes it after five seconds (5000 milliseconds).

Code:
```
function winClose() {
    myWindow.close()
}

myWindow = window.open("", "tinyWindow", 'width=150, height=110')
myWindow.document.write("This window will close automatically after five seconds. Thanks for
your patience")
self.setTimeout('winClose()', 5000)
```

In this example, the **setTimeout** method is used with the **onClick** core attribute in an **input** tag within the **body** element to call a function after five seconds (5000 milliseconds):

```
<html>
<head>
<script language="Javascript">
function displayAlert()
{
alert("The GURU sez hi!")
}
</script>
</head>
<body>
<form>
Click on the button.
<br>
After 5 seconds, an alert will appear.
<br>
<input type="button" onclick="setTimeout('displayAlert()',5000)" value="Click Me">
</form>
</body>
</html>
```

METHOD: Window::confirm

confirm("message")

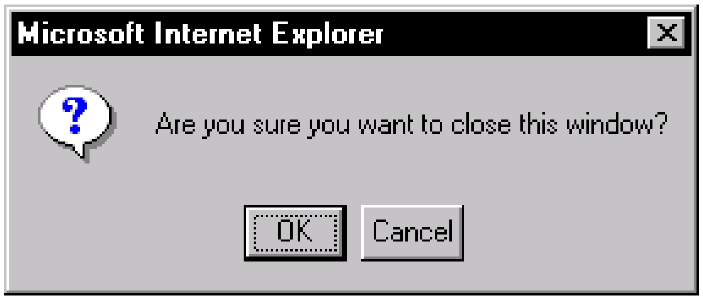
This method brings up a dialog box that prompts the user to select either 'o.k.' or 'cancel', the first returning true and the latter, false.

The following example opens a new window, creates a button in the original window and assigns the closeWindow() function to its onClick event handler. This function prompts the user to confirm the closing of the new window.

Code:

```
<form action="" method="POST" id="myForm">
<input type="Button" name="" value="Close" id="myButton" onClick="closeWindow()">

<script type="" language="JavaScript">

myWindow = window.open("", "tinyWindow", 'toolbar, width=150, height=100')

function closeWindow() {
      myWindow.document.write("Click 'O.K'. to close me and 'Cancel' to leave me open.")
      if (confirm("Are you sure you want to close this window?")) {
            myWindow.close()
      }
}

</script>
</form>
```

Output:


Microsoft Internet Explorer

Are you sure you want to close this window?

METHOD: Window::find

window.find([string[, caseSensitive, backward]])

This method allows the searching of the contents of a window for a specified string. The **caseSensitive** and **backward** arguments are Booleans and to use either of these you must also specify the other. If a search string is not supplied, JavaScript will display a Find dialog box which prompts the user for a string to search for, and also provides the facility to set the other two (**caseSensitive** and **backward**) arguments.

The first example below uses the onClick event handler of the <A> tag to call the findhello() function that searches the contents of the window for the strings "hello" and "goodbye". The results of these searches are displayed, true or false, as JavaScript alerts. The second example shows the "Find" dialog box that is displayed if no search string is supplied.

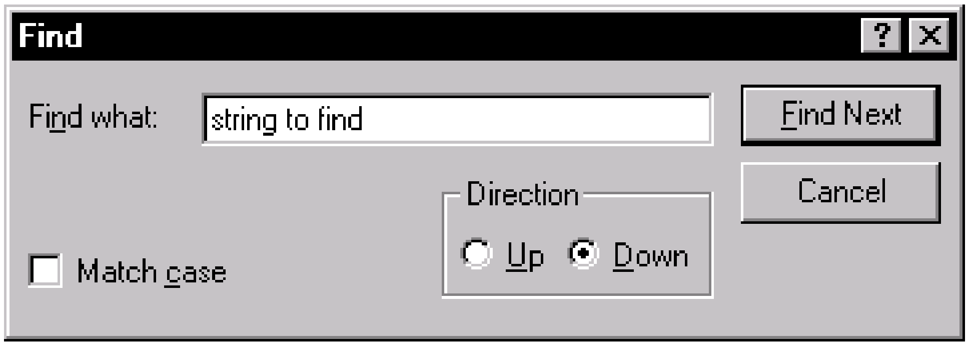
Code:

```
<SCRIPT>
function findhello ()
{
   alert("FIND hello = " + window.find("hello"))
   alert("FIND goodbye = " + window.find("goodbye"))
}
</SCRIPT>

<A HREF="javascript:findhello()">Find hello</A>
hello
```

Code:

```
self.find()
```

Output:



METHOD: Window::forward

window.forward()

Using this method is the same as clicking the browser's Forward button, i.e. it moves to the next URL in the history list of the current top-level window.

The following example creates a button on the page that acts the same as the browser's Forward button.

Code:
```
<input type="button" value="Go back" onClick="window.forward()">
```

Output:

Forward

METHOD: Window::open

window.open(URL, name [, features])

This method is used to open a new browser window. Note that, when using this method with event handlers, you must use the syntax **window.open()** as opposed to just **open()**. Calling just **open()** will, because of the scoping of static objects in JavaScript, create a new document (equivalent to **document.open()**), not a window.

The available parameters are as follows:

URL - this is a string containing the URL of the document to open in the new window. If no URL is specified, an empty window will be created.

name - this is a string containing the name of the new window. This can be used as the 'target' attribute of a <FORM> or <A> tag to point to the new window.

features - this is an optional string that contains details of which of the standard window features are to be used with the new window. This takes the form of a comma-delimited list. Most of these features require yes or no (1 or 0 is also o.k.) and any of these can be turned on by simply listing the feature (they default to yes). Also, if you don't supply any of the feature arguments, all features with a choice of yes or no are enabled; if you do specify any feature parameters, **titlebar** and **hotkeys** still default to yes but all others are no.

Note that many of the values for the **features** parameter are Netscape only. Further, with the exception of **dependent** and **hotkey**, these Netscape only values represent potential sources of security problems and therefore require signed script (and user's permission) if they are to be used.

Details of the available values are given below:

features Value	**Description**
alwaysLowered	When set to yes, this creates a window that always floats below other windows.
alwaysRaised	When set to yes, this creates a window that always floats above other windows.
dependent	When set to yes, the new window is created as a child (closes when the parent window closes and does not appear on the task bar on Windows platforms) of the current window.
directories	When set to yes, the new browser window has the standard directory buttons.
height	This sets the height of the new window in pixels.
hotkeys	When set to no, this disables the use of hotkeys (except security and quit hotkeys) in a window without a menubar.
innerHeight	This sets the inner height of the window in pixels.
innerWidth	This sets the inner width of the window in pixels.
location	When set to yes, this creates the standard Location entry feild in the new browser window.

menubar	When set to yes, this creates a new browser window with the standard menu bar (File, Edit, View, etc.).
outerHeight N	This sets the outer height of the new window in pixels.
outerWidth N	This sets the outer width of the new window in pixels.
resizable	When set to yes this allows the resizing of the new window by the user.
screenX N	This allows a new window to be created at a specified number of pixels from the left side of the screen.
screenY N	This allows a new window to be created at a specified number of pixels from the top of the screen.
scrollbars	When set to yes the new window is created with the standard horizontal and vertical scrollbars, where needed
status	When set to yes, the new window will have the standard browser status bar at the bottom.
titlebar N	When set to yes the new browser window will have the standard title bar.
toolbar	When set to yes the new window will have the standard browser tool bar (Back, Forward, etc.).
width	This sets the width of the new window in pixels.
z-lock N	When set to yes this prevents the new window from rising above other windows when it is made active (given focus).

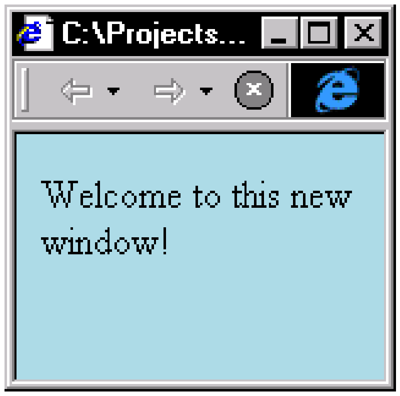
These features may only be used with IE4:

channelmode	sets if the window appears in channel mode.
fullscreen	the new window will appear in full screen.
left	same as screenX, allows a new window to be created at a specified number of pixels from the left side of the screen.
top	same as screenY, allows a new window to be created at a specified number of pixels from the top of the screen.

The following example creates a new window of the specified dimensions complete with toolbar, changes the background color and writes a message to it.

Code:
```
myWindow = window.open("", "tinyWindow", 'toolbar,width=150,height=100')
myWindow.document.write("Welcome to this new window!")
myWindow.document.bgColor="lightblue"
myWindow.document.close()
```

Output:



Welcome to this new window!

METHOD: Window::prompt

window.prompt(message[, defaultInput])

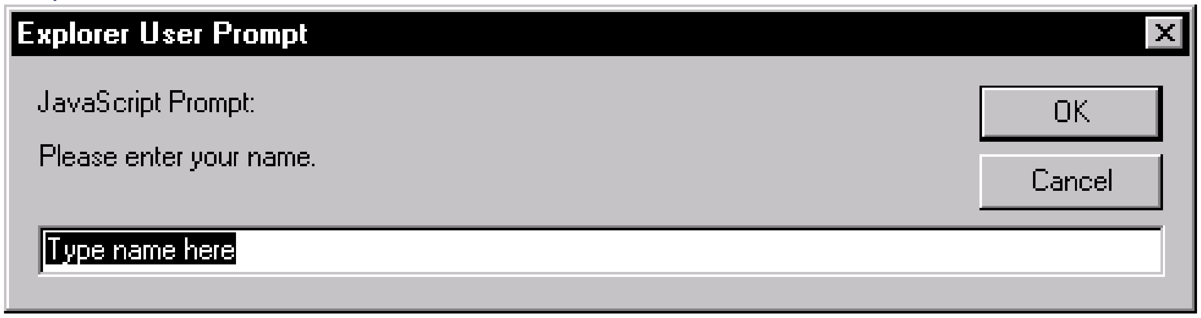
This method displays a dialog box prompting the user for some input. The optional **defaultInput** parameter specifies the text that initially appears in the input field.

The following example prompts the user for their name and then writes a personalized greeting to the page.

Code:

```
<body onload=greeting()>

<script language="JavaScript">
function greeting() {
   y = (prompt("Please enter your name.", "Type name here"))
   document.write("Hello " + y)
}
</script>
```

Output:



```
Explorer User Prompt

JavaScript Prompt:

Please enter your name.

Type name here
```

METHOD: Window::scrollTo

window.scrollTo(xPosition, yPosition)

This method scrolls the contents of a window, the specified co-ordinate becoming the top left corner of the viewable area. Both parameters are integers and they represent the x and y co-ordinates in pixels. This method is only useful where there are areas of the document not viewable within the current viewable area of the window and the visible property of the window's scrollbar must be set to true (enabled).

The following example assigns the toTop() function to the onClick event handler of a button. Note that for this example to have any noticable effect, the button must be placed below the bottom of the default viewable area of the window, i.e. you have to scroll down to be able to see the button.

Code:
```
<script type="" language="JavaScript">

function toTop() {
        self.scrollTo(0, 0)
}

</script>

<form action="" method="POST" id="myForm">
<input type="Button" name="" value="Top" id="myButton" onClick=toTop()>
</form>
```

EVENT HANDLER: onDragDrop

onDragDrop = myJavaScriptCode

Event handler for **Window**.

The **onDragDrop** event handler executes the specified JavaScript code or function on the occurance of a DragDrop event. This is when an object, such as a shortcut or file, is dragged and dropped into the browser window. If the event handler returns true, the browser will attempt to load the dropped item into its window, and if false the drag and drop process is cancelled.

The **onDragDrop** event handler uses the following **Event** object properties.

data - this property returns the URLs of any dropped objects as an Array of Strings.
type - this property indicates the type of event.
target - this property indicates the object to which the event was originally sent.
screenX - the cursor location when the click event occurs.
screenY - the cursor location when the click event occurs.
modifiers - lists the modifier keys (shift, alt, ctrl, etc.) held down when the click event occurs.

EVENT HANDLER: onMouseOut

onMouseOut = myJavaScriptCode

Event handler for **Layer**, **Link**

The **onMouseOut** event handler is used to execute specified Javascript code whenever the user moves the mouse out of an area or link from inside that area or link. If used with an **Area** object, that object must include the HREF attribute within the AREA tag. And if you want to set the **status** or **defaultStatus** properties using the **onMouseOut** event handler, you must return **true** within the event handler. **onMouseOut** uses the following properties of the **Event** object:

type - indicates the type of event.
target - indicates the target object to which the event was sent.
layerX, layerY, pageX, pageY, screenX, screenY
indicate the cursor location at the time of the MouseOut event.

The following example creates an **Image** object. When the user moves the mouse outside the image from within it, the **onMouseOut** event handler changes the picture displayed by calling the 'changeImage' function which (not listed) alters the **src** property of the **Image** object.

Code:
```
<IMG name="myImage" src="images/myPic.jpg" onMouseOut="changeImage()">
```

OBJECT: Area

An **Area** object is a type of **Link** object and shares the same attributes. It defines an area of an image as an image map. When you click on an area, that area's hypertext reference is loaded into the target window. For more information on the **Area** object, see the **Link** object.

EVENT HANDLER: onMouseOver

onMouseOver = myJavaScriptCode

Event handler for **Layer**, **Link**

The **onMouseOver** event handler is used to execute specified Javascript code whenever the user moves the mouse over an area or object from outside that area or object. If used with an **Area** object, that object must include the HREF attribute within the AREA tag. And if you want to set the **status** or **defaultStatus** properties using the **onMouseOver** event handler, you must return **true** within the event handler. **OnMouseOver** uses the following properties of the **Event** object:

type - indicates the type of event.
target - indicates the target object to which the event was sent.
layerX, layerY, pageX, pageY, screenX, screenY
indicate the cursor location at the time of the MouseOver event.

The following example creates an **Image** object. When the user moves the mouse over that image from outside of it, the **onMouseOver** event handler changes the picture displayed by calling the 'changeImage' function which (not listed) alters the **src** property of the **Image** object.

Code:
```
<IMG name="myImage" src="images/myPic.jpg" onMouseOver="changeImage()">
```

EVENT HANDLER: onKeyPress

onKeyPress = myJavaScriptCode

Event handler for **Document**, **Image**, **Link**, **TextArea**.

The **onKeyPress** event handler executes the specified JavaScript code or function on the occurance of a KeyPress event. A KeyPress event occurs when the user presses or holds down a key.

The **onKeyPress** event handler uses the following **Event** object properties.

type - this property indicates the type of event.
target - this property indicates the object to which the event was originally sent.
layerX - the cursor location when the KeyPress event occurs.
layerY - the cursor location when the KeyPress event occurs.
pageX - the cursor location when the KeyPress event occurs.
pageY - the cursor location when the KeyPress event occurs.
screenX - the cursor location when the KeyPress event occurs.
screenY - the cursor location when the KeyPress event occurs.
which - this represents the key pressed as its ASCII value.
modifiers - lists the modifier keys (shift, alt, ctrl, etc.) held down when the KeyPress event occurs.

The following example shows the use of the **onKeyPress** event handler to display a message in the text box.

Code:
```
<body>
<form action="" method="POST" id="myForm" >
<input type="text" name="myText" onKeyPress="changeVal()">

<script type="text/javascript" language="JavaScript">
s1 = new String(myForm.myText.value)

function changeVal() {
     s1 = "You pressed a key"
    myForm.myText.value = s1.toUpperCase()
}

</script>
</form>
</body>
```

EVENT HANDLER: onKeyUp

onKeyUp = myJavaScriptCode

Event handler for **Document**, **Image**, **Link**, **TextArea**.

The **onKeyUp** event handler executes the specified JavaScript code or function on the occurance of a KeyUp event. A KeyUp event occurs when the user releases a key from its depressed position.

The **onKeyUp** event handler uses the following **Event** object properties.

type - this property indicates the type of event.
target - this property indicates the object to which the event was originally sent.
layerX - the cursor location when the KeyUp event occurs.
layerY - the cursor location when the KeyUp event occurs.
pageX - the cursor location when the KeyUp event occurs.
pageY - the cursor location when the KeyUpevent occurs.
screenX - the cursor location when the KeyUp event occurs.
screenY - the cursor location when the KeyUp event occurs.
which - this represents the key released as its ASCII value.
modifiers - lists the modifier keys (shift, alt, ctrl, etc.) held down when the KeyUp event occurs.

The following example shows the use of the **onKeyUp** event handler to display a message in the text box.

Code:
```
<body>
<form action="" method="POST" id="myForm">
<input type="text" name="myText" onKeyUp="changeVal()">

<script type="text/javascript" language="JavaScript">
s1 = new String(myForm.myText.value)

function changeVal() {
     s1 = "You released a key"
    myForm.myText.value = s1.toUpperCase()
}

</script>
</form>
</body>
```

EVENT HANDLER: onMouseDown

onMouseDown = myJavaScriptCode

Event handler for **Button**, **Document** **and** **Link**

The **onMouseDown** event handler is used to execute specified Javascript code whenever the user depresses a mouse button. **onMouseDown** uses the following properties of the **Event** object:

type - indicates the type of event.
target - indicates the target object to which the event was sent.
layerX, layerY, pageX, pageY, screenX, screenY
   indicate the cursor location at the time of the MouseDown event.
which - 1 represents a left mouse click and 3 a right click.
modifiers - lists the modifier keys (shift, alt, ctrl, etc.) held down when the   MouseDown event occurs.

For example, with the following code, the user can click on the word 'green' to alter the background color to green. Depressing the mouse button calls the 'changeColor' function to do this:

Code:
```
<P>Click on <B><A onMouseDown="changeColor()"> green</A></B> to change background
color.</P>
```

EVENT HANDLER: onMouseUp

onMouseUp = myJavaScriptCode

Event handler for **Button**, **Document**, **Link**

The **onMouseUp** event handler is used to execute specified JavaScript code whenever the user releases the mouse button. It uses the following properties of the **Event** object:

type - indicates the type of event.
target - indicates the target object to which the event was sent.
layerx, layerY, pageX, pageY, screenX, screenY
  indicate the cursor position at the time of the MouseUp event.
which - represents 1 for a left-mouse-button up, and 3 for a right-mouse-button up.
modifiers - indicates any modifier keys held down when the MouseUp event occurred.

For example, with the following code, the user can click on the word 'green' to alter the background color to green. Releasing the mouse calls the 'changeColor' function to do this:

Code:
```
<P>Click on <B><A onMouseUp="changeColor()"> green</A></B> to change background color.</P>
```

NOTE:

When an OnMouseUp returns false, such as over an armed link (a MouseDown over a link causes it to become armed), the default action is canceled. This also happens with a MouseUp over an unarmed link. To illustrate this, the following example assumes that a particular link 'lostLink' is no longer available, so when the user clicks on that link, the release of the mouse button calls the 'myAlert' function and causes the link not to be triggered.

Code:
```
<P>Go to <A href="lostLink.html" onMouseUp="myAlert()">MyLink</a>.
</P>
```

EVENT HANDLER

onAbort
onBlur
onChange
onClick
onDblClick
onDragDrop
onError
onFocus
onKeyDown
onKeyPress
onKeyUp
onload
onMouseDown
onMouseMove
onMouseOut
onMouseOver
onMouseUp
onMove
onReset
onResize
onSelect
onSubmit
onUnload

EVENT HANDLER: onload

onload = myJavaScriptCode

Event handler for **Image**, **Layer** and **Window**.

The **onload** event handler executes the specified JavaScript code or function on the occurance of a Load event. A Load event occurs when the browser finishes loading a window or all the frames in a window.

The **onload** event handler uses the following **Event** object properties.

type - this property indicates the type of event.
target - this property indicates the object to which the event was originally sent.
width - when the event is over a window, not a layer, this represents the width of the window.
height - when the event is over a window, not a layer, this represents the height of the window.

The following example shows the use of the **onload** event handler to display a message in the text box.

Code:

```
<body onload = "changeVal()" >
<form action="" method="POST" id="myForm" >
<input type="text" name="myText" >

<script type="text/javascript" language="JavaScript">
s1 = new String(myForm.myText.value)

function changeVal() {
     s1 = "Greetings!"
    myForm.myText.value = s1.toUpperCase()
}

</script>
</form>
</body>
```

EVENT HANDLER: onMouseMove

onMouseMove = myJavaScriptCode

Event handler for no objects as default.

The **onMouseMove** event handler is used to execute specified Javascript code whenever the mouse is moved. **onMouseMove** uses the following properties of the **Event** object: Because MouseMove events occur so often, it is not a default event of any object. To use this event type wth an object you must explicitly set the object to capture MouseMove events.

type - indicates the type of event.
target - indicates the target object to which the event was sent.
layerX, layerY, pageX, pageY, screenX, screenY
indicate the cursor location at the time of the MouseMove event.
which - 1 represents a left mouse click and 3 a right click.
modifiers - lists the modifier keys (shift, alt, ctrl, etc.) held down when the MouseMove event occurs.

EVENT HANDLER: onMove

---

onMove = myJavaScriptCode

Event handler for **Window**

The **onMove** event handler is used to execute specified Javascript code whenever the user or the script moves a window or frame. It uses the following properties of the **Event** object:

type - indicates the type of event.
target - indicates the target object to which the event was sent.
screenX, screenY - indicates the position of the top left corner of the window or   frame.

EVENT HANDLER: onReset

onReset = myJavaScriptCode

Event handler for **Form**

The **onReset** event handler is used to execute specified JavaScript code whenever the user resets a form by cicking a Reset button. It uses the following properties of the **Event** object:

type - indicates the type of event
target - indicates the target object to which the event was sent.

The following example for a possible on-line book club creates a text field for the names of books with the default being the Editor's Book of the Month. The user can at any time reset it to the default which will cause the **onReset** event handler to display a message saying that the text will be reset to the Book of the Month.

Code:
```
<FORM NAME="form1" onReset="alert('Reset to Book of the Month.')">
<P>Select a Book:<BR>
<INPUT TYPE="text" NAME="MonthBook" VALUE="The Joys of JavaScript" SIZE="25"></P>
<P><INPUT TYPE="reset" VALUE="Editor's Choice" NAME="reset1"> </P></FORM>
```

OBJECT: Form

Forms allow us to prompt a user for input using elements such as radio buttons, checkboxes and selection lists. Data gathered in this manner can then be posted to a server for processing. A **form** is created by enclosing HTML controls and other elements within **<form></form>** tags. A page can contain as many forms as required, but they cannot be overlapping or nested (the closing **</form>** tag of a form must precede the opening tag of any subsequent form).

PROPERTIES

action Property
This property specifies the URL address to which the data gathered by the **form** will be submitted. An email address can also be specified using the 'mailto:anybody@anywhere.com' syntax.

Syntax: **object.action = URL**

elements Property
This property is an array containing an object for each element on the **form**. These objects (checkboxes, radio buttons, etc.) are added to the array in the order that they appear in the document's source code.

Syntax: **object.elements**

encoding Property
This property sets the MIME type that is used to encode the data gathered by the elements in a form for submission when using the **post** method. This property initially contains a string reflecting the **enctype** attribute of the **form** tag, but using **encoding** will override this.

Syntax: **object.encoding**

length Property
This property returns the number of elements in a form.

Syntax: **object.length**

method Property
This property is a string specifying how information input in a form is submitted to the server. This should return either 'get', which is the default, or 'post'.

Syntax: **object.method**

name Property
This property sets or returns the name of the form. Initially contains the **name** attribute of the **<form>** tag.

Syntax: **object.name**

target Property
This property sets or returns the target window that responses are sent to after submission of a form.

Syntax: **object.target**

METHODS

handleEvent Method
This method invokes the event handler for the specified event.

Syntax: **object.handleEvent"event"**

reset Method
This method resets the default values of any elements in a form. Emulates the clicking of a Reset button (athough it is not necessary to have a reset button in a form to use this method).

Syntax: **object.reset( )**

submit Method
This method submits a Form. This is the same as clicking a Submit button.

Syntax: **object.submit( )**

EVENT HANDLERS

onReset Event handler
The **onReset** event handler is used to execute specified JavaScript code whenever the user resets a form by cicking a Reset button.

Syntax: **object.onReset="myJavaScriptCode"**

onSubmit Event handler
The **onSubmit** event handler is used to execute specified JavaScript code whenever the user submits a form, and as such, is included within the HTML <form> tag

Syntax: **object.onSubmit="myJavaScriptCode"**

PROPERTIES

OBJECT: Navigator

The **Navigator** object is designed to contain information about the version of Netscape Navigator which is being used. However, it can also be used with Internet Explorer. All of its properties, which are read-only, contain information about different aspects of the browser.

PROPERTIES

appCodeName Property
This property contains a string which specifies the code name of the browser.

Syntax: **navigator.appCodeName**

appName Property
This property is a string that specifies the name of the browser. With a Netscape browser this property contains the string "Netscape", while with an IE explorer it contains "Microsoft Internet Explorer".

Syntax: **navigator.appName**

appVersion Property
This property contains information about the browser version being used. If it is a Netscape browser, it contains the release number, the language used, the platform on which the browser is running, and either the letter I to indicate the international release, or the letter U for the domestic US release which has stronger encryption. e.g., '4.5 [en] (WinNT; I)'. With IE, this property only contains information about the compatible version of Internet Explorer and the platform, such as: '4.0 (compatible; MSIE 4.01; Windows NT)'.

Syntax: **navigator.appVersion**

language property
This Property contains information, usually in the form of two letters, on the language translation of the browser.

Syntax: **navigator.language**

mimeTypes Property
This property is an array of all the MIME (Multipart Internet Mail Extension) types supported by the client.

Syntax: **navigator.mimeTypes**

platform Property
This property contains a string indicating the machine type for which the browser was compiled. e.g., 'Win32' is a 32 bit Windows machine.

Syntax: **navigator.platform**

plugins Property
This property is an array of all the plugins installed on the client. The plugin property also has

its own method: **plugins.refresh**

Syntax: **navigator.plugins**

userAgent Property
This property contains a string representing the value of the user-agent header sent by the client to the server in the http protocol. This information consists of the code name and the version of the browser, and is used by the server to identify the client.

Syntax: **navigator.userAgent**

METHODS

javaEnabled Method
This method tests whether or not Java is enabled returning **true** if it is and **false** if not.

Syntax: **navigator.javaEnabled()**

plugins.refresh Method
This method makes newly-installed plugins available and updates relevent arrays such as the **plugins** array. If you specify the value **true**, this method reloads all open documents containing embedded objects, whereas supplying **false** does not cause this to happen.

Syntax: **navigator.plugins.refresh(true | false)**

preference Method
This method allows a signed script to get and set certain Navigator preferences.

Syntax: **navigator.preference(prefName[, setValue])**

taintEnabled Method
This method determines whether or not data tainting is enabled returning **true** if it is, and **false** if not.

Syntax: **navigator.taintEnabled()**

METHOD: navigator::preference

navigator.preference(prefName[, setValue])

The **preference** method allows a signed script to get and set certain Navigator preferences. These preferences, along with their possible values, are as follows:

autoupdate.enabled
This preference is used to enable SmartUpDate, and its value can be set to either **true** or **false**.

browser.enable_style_sheets
This preference is used to enable style sheet, and its value can be set to either **true** or **false**.

general.always_load_images
This preference is used to automatically load images, and its value can be set to either **true** or **false**.

javascript.enabled
This preference is used to enable JavaScript, and its value can be set to either **true** or **false**.

network.cookie.cookieBehavior
This preference is used to determine how the browser deals with cookies: if its value is set to 0, it accepts all cookies, if 1, it only accepts those cookies that get sent back to the originating server, and if it is set to 2, it'll disable cookies

network.cookie.warnAboutCookies
This preference is used to warn before accepting cookies, and its value can be set to either **true** or **false**.

security.enable_java
This preference is used to enable Java, and its value can be set to either **true** or **false**.

NOTES:

To read a preference requires a **UniversalPreferencesRead** privilege, while setting a preference requires a **UniversalPreferencesWrite** privilege.

OBJECT: Screen

A **Screen** object, automatically created by the JavaScript runtime engine, returns information on the display screen's dimensions and color depth.

PROPERTIES

availHeight Property
This property returns the height of the screen in pixels, minus any permanent or semi-permanent components of the operating system's interface i.e. Windows' Taskbar feature.

Syntax: **screen.availHeight**

availWidth Property
This property returns the width of the screen in pixels, minus any permanent or semi-permanent components of the operating system's interface i.e. Windows' Taskbar feature

Syntax: **screen.availWidth**

colorDepth Property
If a color palette is in use, this property returns its bit depth. If not, the value reflects the **screen.pixelDepth** property.

Syntax: **screen.colorDepth**

height Property
This property returns the height of the display screen.

Syntax: **screen.height**

pixelDepth Property
This property returns the color resolution, in bits per pixel, of the display screen.

Syntax: **screen.pixelDepth**

width Property
This property returns the width of the display screen.

Syntax: **screen.width**

METHODS

The **Screen** object inherits the **watch** and **unwatch** methods of the **Object** object

OBJECT: Array

new Array(arrayLength)

new Array(element0, element1, ..., elementN)

An array is an ordered set of values grouped together under a single variable name created by using an **Array** object constructor. You can create an **Array** literal by specifying the name of the array and the values of all its elements. The following example creates an array of three elements:

Code:
```
cars = new Array("Mercedes", "Ford", "Chrysler")
```

The elements of an array are indexed using their ordinal number, starting with 0. You could, therefore, refer to the second element in the above array ("Ford") as 'cars[1]'. You can specify the number of elements in a new array by using a single numeric parameter with the **Array** constructor.

For example, the following code creates an array of 7 elements:

Code:
```
fruit = new Array(7)
```

If you create an array with a single numeric parameter, that number is stored in the **length** property, and the array doesn't actually have any elements until some are specifically assigned to it. If, however, the parameter is not a number, an array of 1 element is created and that value assigned to it. You can easily increase the size of an array by assigning a value to an element higher than its current length.

NOTE:

If you specify 'language="Javascript1.2"' in the <SCRIPT> tag and use a single numeric parameter with the **Array** constructor, it will be seen as the value of a single element of the array rather than the number of elements you want that array to contain.

PROPERTIES

constructor Property
The **constructor** property contains the function that created an object's prototype.

Syntax: **object.constructor**

index Property
The read-only **index** property for an array created by a regular expression match and containing the zero-based index of that match.

Syntax: **object.index**

input Property
The read-only **input** property for an array created by a regular expression match and containing the original string against which the match was made.

Syntax: **object.input**

length Property
The **length** property holds an unsigned 32 bit integer representing the length of the array. It can be altered independently of the number of elements in the array.

Syntax: **object.length**

prototype Property
The **prototype** property allows the addition of properties to an array.

Syntax: **object.prototype**

METHODS

concat Method
The **concat** method joins two or more **Array** objects producing one new one. The original **Array** objects are unaffected by this but, if one copy of a string or number is altered, it is not reflected in the other, whereas a change to an object reference can be seen in both copies.

Syntax: **Array.concat(arrayName2, arrayName3, ..., arrayNameN)**

join Method
The **join** method is used to join all the elements of an array into a single string separated by a specified string separator (if none is specified, the default is a comma).

Syntax: **Array.join(separator)**

pop Method
The **pop** method is used to remove and return the last element of an array. This affects the length of the array.

Syntax: **Array.pop()**

push Method
The **push** method is used to add one or more elements to an array, returning the new length of it. This affects the length of the array.

Syntax: **Array.push(element1, ..., elementN)**

reverse Method
The **reverse** method, as the name implies, reverses the order of the elements in an array making the first last and the last first. Syntax: **Array.reverse()**

shift Method
The **shift** method removes and returns the first element of an array. This affects the length of the array.

Syntax: **Array.shift()**

slice Method

The **slice** method creates a new array from a selected section of an array.

Syntax: **Array.slice(begin[,end])**

splice Method

The **splice** method is used to add and/or remove elements of an array.

Syntax; **Array.splice(index, howMany, [element1][, ..., elementN])**

sort Method

The **sort** method sorts the elements of an **array**.

Syntax: **Array.sort(compareFunction)**

toSource Method

The **toSource** method is inherited from the **Object** object and returns the source code of the array. For details see the **Object.toSource** method.

Syntax: **Array.toSource()**

toString Method

The **toString** method is inherited from the **Object** object and returns a string representing the specified array and its elements. For more details see the **Object.toString** method.

Syntax: **Array.toString()**

unshift Method

The **unshift** method adds one or more elements to the beginning of an array and returns the new length.

Syntax: **Array.unshift(element1,..., elementN)**

valueOf Method

The **valueOf** method is inherited from the **Object** object and returns a primitive value for a specified array. For details see the **Object.valueOf** method.

Syntax: **Array.valueOf()**

METHOD: Array::concat

Array.concat(arrayName2, arrayName3, ..., arrayNameN)

The **concat** method joins two or more **Array** objects producing one new one. The original **Array** objects are unaffected by this but, if one copy of a string or number is altered, it is not reflected in the other, whereas a change to a referenced object can be seen in both **Array** objects. The following example adds the elements of the array 'cars' onto the array 'trees' on to the 'Cats' array:

Code:
```
Cats.concat(trees, cars)
```

METHOD: Array::join

Array.join(separator)

The **join** method is used to join all the elements of an array into a single string separated by a specified string separator (if none is specified, the default is a comma).

The following example produces a string of all the elements of the array 'cars' separated by a plus sign (+):

Code:
```
cars.join(" + ")
```

Output:
```
Mercedes + Ford + Chrysler
```

METHOD: Array::pop

Array.pop()

The **pop** method is used to remove and return the last element of an array. This affects the length of the array. The following example creates an array called 'cars' with the listed elements, and the **pop** method then removes and returns the last element "chrysler" leaving just the two elements "Mercedes" and "Ford" in the array:

Code:

```
cars = ["Mercedes", "Ford", "Chrysler"]
document.write(cars.pop())
```

Output:

```
Chrysler
```

METHOD: Array::push

Array.push(element1, ..., elementN)

The **push** method is used to add one or more elements to an array, returning the new length of it. This affects the length of the array.

The following example creates an array 'trees' of two elements and then adds two more using the **push** method.

Code:
```
trees = ["oak", "ash"]
document.write(trees.push("beech", "pear"))
```

Output:
```
4
```

METHOD: Array::shift

Array.shift()

The **shift** method is used to remove and return the first element of an array. This affects the length of the array. The following example creates an array called 'cars' with the listed elements, and the **shift** method then removes and returns the first element "Mercedes" leaving just the two elements "Ford" and "Chrysler" in the array:

Code:
```
cars = ["Mercedes", "Ford", "Chrysler"]
document.write(cars.shift())
```

Output:
```
Mercedes
```

METHOD: Array::slice

Array.slice(begin[,end])

The **slice** method creates a new array from a selected section of an array. The original array is unaffected by this but, if a string or number in one array is altered, it is not reflected in the other, whereas a change to a referenced object can be seen in both **Array** objects. The **slice** method uses the zero-based array index to determine the section out of which to create the new array. It extracts up to, but not including, the 'end' element (if no 'end' is specified, the default is the very last element). The following code creates an array called 'trees' and then displays a 'slice' of it:

Code:
```
trees = ["oak", "ash", "beech", "maple", "sycamore"]
document.write(trees.slice(1,4))
```

Output:
```
ash,beech,maple
```

If you use a negative index for the 'end', this specifies an element so many places from the end. Continuing with the above example, the following code would display the second through the third to last elements of the array:

Code:
```
trees = ["oak", "ash", "beech", "maple", "sycamore"]
document.write(trees.slice(1,-2))
```

Output:
```
ash,beech
```

METHOD: Array::splice

Array.splice(index, howMany, [element1][, ..., elementN])

The **splice** method is used to add and/or remove elements of an array returning an array of the elements removed. You first need to specify the index at which you wish to start removing elements, and the number to remove. (if 'howMany' is 0, you should specify at least 1 element to add). The following code creates an array 'cars' and then displays two elements starting with 'cars[1]':

Code:
```
cars = ["Mercedes", "Ford", "Chrysler", "Honda", "Volvo"]
document.write(cars.splice(1,2))
```

Output:
```
Ford,Chrysler
```

You can also include optional new elements to replace the ones removed. Expanding on the previous example, the following code would create an array consisting of the two extracted elements "Ford" and "Chrysler", but replace them with "Citreon" in the original array:

Code:
```
cars = ["Mercedes", "Ford", "Chrysler", "Honda", "Volvo"]
removed_cars = cars.splice(1, 2, "Citreon")
document.write(removed_cars + "<BR>")
document.write(cars)
```

Output:
```
Ford,Chrysler
Mercedes,Citreon,Honda,Volvo
```

METHOD: Array::sort

Array.sort(compareFunction)

The **sort** method sorts the elements of an array. If no **compareFunction** argument is supplied, all the elements are converted into strings and sorted lexicographically (i.e. in dictionary order). This means, for example, that 30 would come before 4. The following example is a straight-forward sort of an array of names:

Code:
```
names = ["John", "Andrea", "Charlie", "Sam", "Kate"]
sorted_names = names.sort()
document.write(sorted_names)
```

Output:
```
Andrea,Charlie,John,Kate,Sam
```

By including a **compareFunction** argument, you can define the sort order. Two array elements are sorted according to the return value of the compare function: if it is 0, the order of the two elements remains unchanged; if it is greater than 0, the first of the two elements is sorted to a higher index than the second; and if it is less than 0, the second element is sorted to a higher index than the first. The following code creates an array called 'trees' and then, using the user-defined function 'reverseSort', displays the elements sorted in reverse order:

Code:
```
trees = ["oak", "ash", "beech", "maple", "sycamore"]
function reverseSort(a, b)
{
  if(a > b)
    return -1
  if(a < b)
    return 1
  return 0
}
document.write(trees.sort(reverseSort))
```

Output:
```
sycamore,oak,maple,beech,ash
```

If two numbers are compared, the **compareFunction** simply needs to subtract the second from the first number:

Code:
```
ages = [30, 25, 47, 19, 21, 8]
function sortNumbers(a, b) { return a - b}
document.write(ages.sort(sortNumbers))
```

Output:
```
8,19,21,25,30,47
```

METHOD: Array::unshift

Array.unshift(element1,..., elementN)

The **unshift** method adds one or more elements to the beginning of an array and returns the new length. The following code first creates an array called 'trees' and then uses the **unshift** method to add two new elements to the beginning of it, returning the new length of the array. Finally, the third line of code displays all the elements of the altered array:

Code:
```
trees = ["beech", "maple", "sycamore"]
document.write(trees.unshift("oak", "ash"))
document.write("<BR>" + trees)
```

Output:
```
5
oak,ash,beech,maple,sycamore
```

OBJECT: Boolean

new Boolean(value)

The **Boolean** object is an object wrapper for a Boolean value and is constructed with the above Boolean constructor. If there is no initial value or if it is 0, -0, **null**, **false**, **NaN**, **undefined**, or the empty string (""), the initial value is **false**. Otherwise, even with the string "false", it is true. So, all the following objects have an initial value of **false**:

```
x = new Boolean()
x = new Boolean(0)
x = new Boolean(-0)
x = new Boolean(null)
x = new Boolean(false)
x = new Boolean(NaN)
x = new Boolean(undefined)
x = new Boolean("")
```

...whereas in the following examples the Boolean object 'x' has an initial value of **true**:

```
myBool = new Boolean(false)
x = new Boolean(myBool)

x = new Boolean("false")
```

Any **Boolean** object that is passed to a conditional statement (except those with an initial value of **null** or **undefined**) evaluates to true. So, for instance, the conditional statement in the following code evaluates to **true**.

Code:
```
x = new Boolean(false)
if(x)
```

However, this does not apply to Boolean primitives, and the conditional statement in the following code evaluates to **false**.

Code:
```
x = false
if(x)
```

NOTE:

In JavaScript 1.3 and later versions, don't use a **Boolean** object instead of a Boolean primitive, nor should you use a **Boolean** object to convert a non-Boolean value to a Boolean one. To do so use **Boolean** as a function. For example, the following converts the expression 'a+b' to a Boolean value:

Code:
```
x = Boolean(a+b)
```

PROPERTIES

constructor property
This property specifies the function that created the object's prototype. See also the **Object.constructor** property.

Syntax: **object.constructor**

prototype property
This property represents the prototype for this object and allows you to add methods and properties of your own. See also the **Function.prototype** property.

Syntax: **object.prototype**

METHODS

toSource method
This method, which is usually called internally by JavaScript, returns a string representing the source code of the object. It overrides the **Object.toSource** method.

Syntax: **object.toSource()**

toString method
This method converts a **Boolean** object to a string representing its value: i.e. either "true" or "false", and is called by JavaScript automatically whenever a **Boolean** object is used in a situation requiring a string. This method overrides the **Object.toString** method.

Syntax: **object.toString()**

valueOf method
This method, which is usually called internally by JavaScript, returns a primitive value (either "true" or "false") for the **Boolean** object. It overrides the **Object.valueOf** method.

Syntax: **object.valueOf()**

NOTE:

The **Boolean** object also inherits the **watch** and **unwatch** methods from the **Object** object.

OBJECT: Number

new Number(value)

The **Number** object is an object wrapper for primitive numeric values, allowing for their manipulation. To create a **Number** object use the **Number** constructor above. The following example creates a **Number** object of the numeric value 5:

Code:
```
five = new Number(5)
```

The main reason for doing this is to be able to use the constant properties for the **Number** object, although you can create one in order to add properties to it. You can also convert any object to a number by using the **Number** function.

PROPERTIES

constructor property
This property specifies the function that created the object's prototype. See also the **Object.constructor** property.

Syntax: **object.constructor**

MAX_VALUE property
This property represents the largest value possible in JavaScript. It is a static property and hence always referred to as **Number.MAX_VALUE**, and has a value of approximately 1.79769e+308. Numbers larger than this are represented as infinity.

Syntax: **Number.MAX_VALUE**

MIN_VALUE property This property represents the smallest positive number possible in JavaScript, and as a static property is always referred to as **Number.MIN_VALUE**. Its value is 5e-324, and any value smaller than that is converted to 0.

Syntax: **Number.MIN_VALUE**

NaN property
This read-only property represents the special value Not-a-Number, and is always unequal to any other number (including 0) and to NaN itself. As a static property, it is always referred to as **Number.NaN**.

Syntax: **Number.NaN**

NEGATIVE_INFINITY property
This static, read-only property is a special value representing negative infinity, which is returned on overflow.

Syntax: **Number.NEGATIVE_INFINITY**

POSITIVE_INFINITY property
This static, read-only property is a special value representing infinity, which is returned on overflow.

Syntax: **Number.POSITIVE_INFINITY**

prototype property
This property represents the prototype for this object and allows you to add methods and properties of your own. See also the **Function.prototype** property.

Syntax: **object.prototype**

METHODS

toSource method
This method, which is usually called by JavaScript behind the scenes, returns a string representing the source code of the **Number** object. This method overrides the **Object.toSource** method.

Syntax: **object.toSource()**

toString method
This method returns a string representing the **Number** object, and is called by JavaScript whenever the code requires a string value. The optional 'radix' parameter is an integer between 2 and 36 which specifies the base to be used when representing numeric values. This method overrides the **Object.toString** method.

Syntax: **object.toString([radix])**

valueOf method
This method, which is usually called by Javascript behind the scenes, returns the primitive value of a **Number** object as a number data type. This method overrides the **Object.valueOf** method.

Syntax: **object.valueOf()**

NOTE:

The **Number** object also inherits the **watch** and **unwatch** methods from the **Object** object.

PROPERTY: Number::NEGATIVE_INFINITY
PROPERTY: Number::POSITIVE_INFINITY

NEGATIVE_INFINITY
This static, read-only property is a special value representing negative infinity, which is returned on overflow.

Syntax: **Number.**NEGATIVE_INFINITY

The **NEGATIVE_INFINITY** property behaves in the following ways:

Any positive value (including **POSITIVE_INFINITY**) multiplied by **NEGATIVE_INFINITY** is **NEGATIVE_INFINITY**.

Any negative number (including **NEGATIVE_INFINITY**) multiplied by **NEGATIVE_INFINITY** is **POSITIVE_INFINITY**.

Zero multiplied by **NEGATIVE_INFINITY** is **NaN**.

NaN multiplied by **NEGATIVE_INFINITY** is **NaN**.

NEGATIVE_INFINITY divided by any negative number except **NEGATIVE_INFINITY** is **POSITIVE_INFINITY**.

NEGATIVE_INFINITY divided by any positive number except **POSITIVE_INFINITY** is **NEGATIVE_INFINITY**.

NEGATIVE_INFINITY divided by either **NEGATIVE_INFINITY** or **POSITIVE_INFINITY** is **NaN**.

Any number divided by **NEGATIVE_INFINITY** is Zero.

POSITIVE_INFINITY
This static, read-only property is a special value representing infinity, which is returned on overflow.

Syntax: **Number.**POSITIVE_INFINITY

The **POSITIVE_INFINITY** property behaves in the following ways:

Any positive number (including **POSITIVE_INFINITY**) multiplied by **POSITIVE_INFINITY** is **POSITIVE_INFINITY**.

Any negative number (including **NEGATIVE_INFINITY**) multiplied by **POSITIVE_INFINITY** is **NEGATIVE_INFINITY**.

Zero multiplied by **POSITIVE_INFINITY** is **NaN**.

NaN multiplied by **POSITIVE_INFINITY** is **NaN**.

POSITIVE_INFINITY divided by any negative number, except **NEGATIVE_INFINITY**, is **NEGATIVE_INFINITY** .

POSITIVE_INFINITY divided by any positive number, except **POSITIVE_INFINITY**, is **POSITIVE_INFINITY**.

POSITIVE_INFINITY divided by either **NEGATIVE_INFINITY** or **POSITIVE_INFINITY** is **NaN**.

Any number divided by **POSITIVE_INFINITY** is Zero.

OBJECT: String

A **String** object, created using the **String** constructor, represents a series of characters in a string. The string can be enclosed between single or double quotes.

The following code shows the use of the **String** constructor to create a **String** object called 'myString'.

Code:
```
myString = new String("This is a string object")
```

PROPERTIES

constructor Property
This property returns a reference to the function that created the **String** object's prototype.

Syntax: **object.constructor**

length Property
This property returns the length of the string.

Syntax: **object.length**

prototype Property
This property is used to add properties and methods to an object.

Syntax: **object.prototype**

METHODS

anchor Method
This method is used to create an HTML anchor.

Syntax: **object.anchor("name")**

big Method
This method displays the string using a big font, as if contained within HTML <BIG></BIG> tags.

Syntax: **object.big( )**

blink Method
This method makes the displayed string blink, as if contained within HTML <BLINK></BLINK> tags.

Syntax: **object.blink( )**

bold Method
This method displays the string using a bold font, as if contained within HTML <B></B> tags.

Syntax: **object.bold()**

charAt Method
This method returns a character from a string by referring to its index within that string.

Syntax: **object.charAt(index)**

charCodeAt Method
This method returns a character's Unicode value from a string by referring to its index within that string.

Syntax: **object.charCodeAt(index)**

concat Method
This method joins the text contained in one string with the text from other specified strings and returns a new string.

Syntax: **object.concat(strName2, strName3....strName[n])**

fixed Method
This method displays the string using a fixed-pitch font, as if contained within HTML <TT></TT> tags.

Syntax: **object.fixed()**

fontcolor Method
This method displays the string using a specified color, as if contained within HTML <FONT COLOR="color"></FONT> tags.

Syntax: **object.fontcolor("color")**

fontsize Method
This method displays the string using a specified font size, as if contained within HTML <FONT SIZE="fontsize"></FONT> tags.

Syntax: **object.fontsize("fontsize")**

fromCharCode Method
This method takes the specified Unicode values and returns a string.

Syntax: **String.fromCharCode(num1,....,numN)**

indexOf Method
When called from a **String** object, this method returns the index of the first occurance of the specified **searchValue** argument, starting from the specified **fromIndex** argument.

Syntax: **object.indexOf(searchValue,[fromIndex])**

italics Method
This method displays the string using italics, as if contained within HTML <I></I> tags.

Syntax: **object.italics()**

lastIndexOf Method
When called from a **String** object, this method returns the index of the last occurance of the specified **searchValue** argument, searchng backwards from the specified **fromIndex** argument.

Syntax: **object.lastIndexOf(searchValue,[fromIndex])**

link Method
This method is used to create an HTML hyperlink in a document.

Syntax: **object.link("targetURL")**

match Method
This method is used to match a specified regular expression against a string.

Syntax: **object.match(regexp)**

replace Method
This method is used to match a specifed regular expression against a string and replace any match with a new substring.

Syntax: **object.replace(regexp, newSubString)**

search Method
This method is used to search for a match between a regular expression and the specified string.

Syntax: **object.search(regexp)**

slice Method
This method is used to 'slice' a section of a string and return a new string containing that section.

Syntax: **object.slice(startSlice, endSlice)**

small Method
This method displays the string using a small font, as if contained within HTML <SMALL></SMALL> tags.

Syntax: **object.smal()**

split Method
This method splits a string into substrings and creates an array containing the resulting substrings.

Syntax: **object.split([separator][, limit])**

strike Method
This method displays the string using struck-out text, as if contained within HTML <STRIKE></STRIKE> tags.

Syntax: **object.strike()**

sub Method
This method displays the string as subscript text, as if contained within HTML <SUB></SUB> tags.

Syntax: **object.sub()**

substr Method
This method extracts the characters from a string beginning at the specified start index for the specified number of characters.

Syntax: **object.substr(start[, length])**

substring Method

This method returns the characters in a string between two specified indices as a substring.

Syntax: **object.substring(indexA, indexB)**

sup Method
This method displays the string as superscript text, as if contained within HTML <SUP></SUP> tags.

Syntax: **object.sup()**

toLowerCase Method
This method is used to convert the characters in a string to lower case.

Syntax: **object.toLowerCase( )**

toSource Method
This method is used to return a string that represents the source code of the object. Note that the source code of native JavaScript objects will not be available using this method, i.e. the value returned if applying this method to the generic **String** object will be [native code] whereas applying it to an instance of a **String** object created in your code will return that code.

Syntax: **object.toSource( )**

toString Method
This method is used to return a string representation of an object, i.e. **myString.toString()** will simply return the contents of myString.

Syntax: **object.toString( )**

toUpperCase Method
This method is used to convert characters in a string to upper case.

Syntax: **object.toUpperCase( )**

valueOf Method
This method returns the primitive value of a **String** object as a string datatype. The value returned using this method is identical to using **String.toString**.

Syntax: **object.valueOf**

METHOD: String::(various formatting methods)

object.'formatting_method'( )

There are various methods available that allow you to perform formatting alterations to the contents of your **String** object when written to a document. These methods are:

big, **blink**, **bold**, **fixed**, **fontcolor**, **fontsize**, **italics**, **small**, **strike**, **sub** and **sup**.

The following code writes the contents of the "myString" **String** object to the document with a different formatting method applied each time it is written.

Code:
```
myString = new String("Formatting methods")
document.write ("<p>big() = " + myString.big())
document.write ("<br>blink() = " + myString.blink())
document.write ("<br>bold() = " + myString.bold())
document.write ("<br>fixed() = " + myString.fixed())
document.write ("<br>fontcolor("red") = " + myString.fontcolor("red"))
document.write ("<br>fontsize("5") = " + myString.fontsize("5"))
document.write ("<br>small() = " + myString.small())
document.write ("<br>strike() = " + myString.strike())
document.write ("<br>sub() = " + myString.sub())
document.write ("<br>sup() = " + myString.sup())
```

Output:

big() = Formatting methods
blink() = Formatting methods
bold() = **Formatting methods**
fixed() = Formatting methods
fontcolor("red")= Formatting methods
fontsize("5") = Formatting methods
small() = Formatting methods
strike() = ~~Formatting methods~~
sub() = $_{\text{Formatting methods}}$
sup() = $^{\text{Formatting methods}}$

NOTE: blink() may not work on some browsers

METHOD: String::charAt

object.charAt(index)

This method returns a character from a string by referring to its index within that string. The characters in a string are indexed from left to right with the first character indexed as 0 and the last as **String.length** - 1.

The following code reads the characters from the string at the specified indicies and writes them to the document.

Code:
```
myString = new String("charAt method demonstration.")
document.write (myString.charAt(26))
document.write (myString.charAt(8))
document.write (myString.charAt(2))
document.write (myString.charAt(5))
```

Output:
```
neat
```

METHOD: String::charCodeAt

object.charCodeAt(index)

This method returns a character's Unicode value from a string by referring to its index within that string. The characters in a string are indexed from left to right with the first character indexed as 0 and the last as **String.length** - 1.

The following code reads a character from the string at the specified index and writes its Unicode value to the document.

Code:
```
myString = new String("charCodeAt method demonstration.")
document.write (myString.charCodeAt(2))
```

Output:
```
65
```

METHOD: String::concat

object.concat(strName2, strName3....strName[n])

This method joins the text contained in one string with the text from other specified strings and returns a new string.

The following code combines the text contained in two specified strings and writes the concatenated string to the document.

Code:
```
myString1 = new String("This demonstrates the ")
myString2 = new String("concat method.")
document.write(myString1.concat(myString2))
```

Output:
```
This demonstrates the concat method.
```

METHOD: String::fromCharCode

String.fromCharCode(num1,..........,numN)

This method takes the specified Unicode values and returns a string (but not a **String** object). Note that this method is a static method of String. Therefore, it is not used as a method of a **String** object that you have created. The syntax is always **String.fromCharCode()** as opposed to **myStringObject.fromCharCode()**.

The following code takes the specified Unicode values and writes the resulting string to the document.

Code:

```
document.write (String.fromCharCode(67,65,66))
```

Output:

```
CAB
```

METHOD: String::indexOf

object.indexOf(searchValue,[fromIndex])

When called from a **String** object, this method returns the index of the first occurance of the specified **searchValue** argument, starting from the specified **fromIndex** argument. If the searchValue is not found, a value of -1 is returned. Characters in the string are indexed from left to right with the first character indexed as **0** and the last as **String.length-1**. Note that this method is case sensitive as shown in the example below.

The following code uses the **indexOf** method to find the index values of three different character strings within the **myString** object. Note that the third example returns **-1** because the case of one of the characters does not match.

Code:
```
myString = new String("DevGuru.com")
document.writeln(myString.indexOf("Guru"))
document.writeln("<br>" + myString.indexOf("com"))
document.writeln("<br>" + myString.indexOf("Com"))
```

Output:
```
3
8
-1
```

METHOD: String::lastIndexOf

object.lastIndexOf(searchValue,[fromIndex])

When called from a **String** object, this method returns the index of the last occurrence of the specified **searchValue** argument, searchng backwards from the specified **fromIndex** argument. If the searchValue is not found, a value of -1 is returned. Chaacters in the string are indexed from left to right with the first character indexed as **0** and the last as **String.length-1**. Note that this method is case sensitive as shown in the example below.

The following code uses the **lastIndexOf** method to find the index values of four different character strings within the **myString** object. Note that the third example returns **-1** because there is no occurrence of the specified string before index value **6**, and the fourth example returns **-1** because the case of one of the characters does not match.

Code:
```
myString = new String("DevGuru.com")
document.writeln(myString.lastIndexOf("u"))
document.writeln("<br>" + myString.lastIndexOf("u",5))
document.writeln("<br>" + myString.lastIndexOf("com",6))
document.writeln("<br>" + myString.lastIndexOf("Com"))
```

Output:
```
6
4
-1
-1
```

METHOD: String::link

object.link("targetURL")

This method is used to create an HTML hyperlink in a document.

The following code writes the contents of the String object to the document as an HTML hyperlink that takes the user to the specified "targetURL". This has identical results as using the HTML code:
<A HREF="http//www.devguru.com">DevGuru.com</A>

Code:

```
myString = new String("DevGuru.com")
document.write (myString.link(www.devguru.com))
```

METHOD: String::match

object.match(regexp)

This method is used to match a specifed regular expression against a string. If one or more matches are made, an array is returned that contains all of the matches. Each entry in the array is a copy of a string that contains a match. If no match is made, a null is returned.

To perform a global match you must include the 'g' (global) flag in the regular expression and to perform a case-insensitive match you must include the 'i' (ignore case) flag.

The following code uses the match method to search the characters in the myString string for a match with the specified regular expression and displays the resulting string in the browser. Note that the 'i' flag is used to make the search case-insensitive.

Code:

```
myString = new String("DevGuru.com")
myRE = new RegExp("guru", "i")
results = myString.match(myRE)
for(var i =0: i < results.length; i++)
  {
    document.write(results[i])
  }
```

Output:

```
Guru
```

METHOD: String::replace

---

object.replace(regexp, newSubStr)
object.replace(regexp, function)

This method is used to match a specifed regular expression against a string and replace any match with a new substring. The **newSubStr** argument can include certain **RegExp** properties. These are: **$1** thru **$9**, **lastMatch**, **lastParen**, **leftContext** and **rightContext** (for details on the **RegExp** object's properties, go here). To perform a global match include the '**g**' (global) flag in the regular expression and to perform a case-insensitive match include the '**i**' (ignore case) flag.

The second argument can also be a function which is invoked after the match is performed and the result of which is used as the replacement string.

The following code uses the **replace** method to alter 'DevGuru.com' in the original string to the full URL for the DevGuru website and then uses the **String.link** method to provide a hyperlink to the site.

Code:
```
myString = new String("Go to DevGuru.com")
rExp = /devguru.com/gi;
newString = new String ("http://www.devguru.com")
results = myString.replace(rExp, newString.link("http://www.devguru.com"))
document.write(results)
```

Output:
```
Go to http://www.devguru.com
```

METHOD: String::search

object.search(regexp)

This method is used to search for a match between a regular expression and the specified string. If a match is found, the search method returns the index of the regular expression within the string. If no match is found, a value of -1 is returned.

The first example of code below uses the **search** method to look for a match between the regular expression 'rExp' and the characters in the 'myString' string object and displays the resulting value in the browser. In the second example, removing the 'i' (ignore case) flag from the regular expression causes the value of the search to be returned as -1

Code:
```
myString = new String("Go to DevGuru.com")
rExp = /devguru.com/gi;
results = myString.search(rExp)
document.write(results)
```

Output:
```
6
```

```
myString = new String("Go to DevGuru.com")
rExp = /devguru.com/g;
results = myString.search(rExp)
document.write(results)
```

Output:
```
-1
```

METHOD: String::slice

object.slice(startSlice[, endSlice])

This method is used to 'slice' a section of a string and return a new string containing that section. The section of the string that is sliced begins at the specified index **startSlice** and ends at the index before **endSlice**, i.e. up to but not including **endSlice**. A negative value can be given for endSlice, in which case the section of the string that is extracted ends at that value from the end of the original string. See the example for details.

Both examples below return the same string. The first uses a positive value for the **endSlice** parameter and the second a negative value.

Code:
```
myString = new String("Go to DevGuru.com")
slicer=myString.slice(1,8)
document.write(slicer)
```

Output:
```
o to De
```

```
myString = new String("Go to DevGuru.com")
slicer=myString.slice(1,-9)
document.write(slicer)
```

Output:
```
o to De
```

METHOD: String::split

object.split [delimiter]

This method is used to 'split' a string into an array of substrings. The array numbering starts at zero.

The optional **delimiter** argument is the character, regular expression, or substring that is used to determine where to split the string. The **delimiter** value is not returned as part of the array of substrings.

If the argument is a character or substring, it must be enclosed with in a pair of double quotes. If the argument is a regular expression, do not enclose it with double quotes. If no **delimiter** argument is provided, the string is not split. If the empty string, "", is used as the **delimiter**, the string is split between each character in the string.

Code:

```
myString = new String("A,B,C,D")
splitString = myString.split(",")
document.write("<BR>" + splitString[0])
document.write("<BR>" + splitString[1])
document.write("<BR>" + splitString[2])
document.write("<BR>" + splitString[3])
```

Output:

```
A
B
C
D
```

OBJECT: Hidden

A **Hidden** object provides a text object on a form that is hidden from the user and is created with every instance of an HTML <INPUT> tag with the type attribute set to 'hidden'. A **Hidden** object is used to pass name/value pairs when the form is submitted. These objects are then stored in the elements array of the parent form and accessed using either the name defined within the HTML tag or an integer (with '0' being the first element defined, in source order, in the specified form).

The following example creates a button that, when clicked, displays the **Hidden** object's value in an alert box.

Code:
```
<form action="" method="POST" id="myForm">
<input type="Hidden" name="objHidden" value="" id="objHidden">
<input type="Button" name="" value="Get hidden value" id="myButton" onClick=getHidden()>

<script type="" language="JavaScript">
myForm.objHidden.value=("Hidden object example")

function getHidden() {
     y=myForm.objHidden.value
    self.alert("The Hidden object's value is : " + y)
}

</script> </form>
```

PROPERTIES

form Property
This property returns a reference to the parent **Form** of the **Hidden** object.

Syntax: **object.form**

name Property
This property sets or returns the value of the **Hidden** object's **name** attribute.

Syntax: **object.name**

type Property
Every element on a form has an associated **type** property. In the case of a **Hidden** object, the value of this property is always "hidden".

Syntax: **object.type**

value Property
This property sets or returns the **Hidden** object's **value** attribute. This is the text that is held in the **Hidden** object until the form is submitted.

Syntax: **object.value**

METHODS

The **Hidden** object inherits the watch and unwatch methods of the Object object.

PROPERTY: RegExp::lastIndex

This property is the index at which to start the next match, but is only set if the regular expression uses the 'g' flag to specify a global search. It consists of an integer specifying the index (counting from the beginning of the string and including all alphanumeric and non-alphanumeric characters) of the first character after the last match. So, for example, the following code searches for an occurrence of the substring 'ships' in the string "the hardships of traveling", matches the word 'hardships', and then displays the number 14, which is the index of the letter 'o' of 'of'. Note that the regular expression includes the white (blank) space character \s.

Code:
```
myRexp = /ships*\s/g
myRexp.exec("the hardships of traveling")
document.write(myRexp.lastIndex)
```

Output:
```
14
```

If the value of the **lastIndex** property is greater than the length of the string, then both the **test** and **exec** methods will fail, and the **lastIndex** property will be set to 0. For example, the following code will match the final 'ing' of 'traveling' and set the **lastIndex** property to 26 (one more than the index of the last character of the string). If you then immediately run the match again, it will fail and the **lastIndex** property will be set to 0.

Code:
```
myRexp = /ing/g
myRexp.exec("the hardships of traveling")
```

If the **lastIndex** property is the same as the length of the string, and provided the regular expression doesn't match an empty string (by using '?'), then there will be no match and the **lastIndex** property will be set to 0. If, however, the regular expression does match an empty string, then the regular expression will match the empty string at **lastIndex**.

PROPERTY: RegExp::lastParen

RegExp.lastParen

This property contains the last matched parenthesized substring (if any), and as a static property is always refered to using **RegExp.lastParen**. For example, the following code uses a regular expression containing three parenthesized substrings to search for a match in the string "the fisherman's tale". After successfully matching the substring 'sherm', the **lastParen** property will hold the value 'r', which is the match made by the last parenthesized substring (any consonant).

Code:
```
rexp = /([^aeiou\s]){2}([aeiou])+([^aeiou\s]){2}/
rexp("the fisherman's tale")
```

Output:
```
r
```

PROPERTY: RegExp::leftContext

RegExp.leftContext

This property is the substring upto the character most recently matched (i.e. everything that comes before it), and as a static property, is always used as **RegExp.leftContext**. Consider the following code. The regular expression consists of one or more vowels. The code searches the string "the fisherman" and matches the 'e' in 'the' printing the substring to the left of it: namely 'th'. Then it searches the same string again from where it ended the previous search (see the **lastIndex** property), this time matching the 'i' of 'fisherman' and printing the preceding substring 'the f'.

Code:

```
rexp = /[aeiou]+/g
rexp("the fisherman")
document.write(RegExp.leftContext)
rexp("the fisherman")
document.write("<BR>" + RegExp.leftContext)
```

Output:

```
th
the f
```

NOTE:

The regular expression in the above example uses the flag 'g' to indicate a global search. If it wasn't there, the second search in the above example would start at the beginning of the string producing exactly the same match as the first search: namely 'th'.

PROPERTY: RegExp::rightContext

RegExp.rightContext

This property is the substring following the character most recently matched (i.e. everything that comes after it), and as a static property, is always used as **RegExp.rightContext**. Consider the following code. The regular expression consists of one or more vowels. The code searches the string "the fisherman" and matches the 'e' in 'the' printing the substring to the right of it: namely 'fisherman'. Then it searches the same string again starting from where it ended in the previous search (see the **lastIndex** property), this time matching the 'i' of 'fisherman' and printing the following substring 'sherman'.

Code:

```
rexp = /[aeiou]+/g
rexp("the fisherman")
document.write(RegExp.rightContext)
rexp("the fisherman")
document.write("<BR>" + RegExp.rightContext)
```

Output:

```
fisherman
sherman
```

NOTE:

The regular expression in the above example uses the flag 'g' to indicate a global search. If it wasn't there, the second search in the above example would start at the beginning of the string producing exactly the same match as the first search: namely 'fisherman'.

EVENT HANDLER: onSubmit

---

onSubmit = myJavaScriptCode

Event handler for **Form**

The **onSubmit** event handler is used to execute specified JavaScript code whenever the user submits a form, and as such, is included within the HTML <FORM> tag. The **onSubmit** event handler uses the following properties of the **Event** object:

type - indicates the type of event.
target - indicates the target object to which the event was sent.

In the following example, the **onSubmit** event handler calls the 'submitEvent' function:

Code:
```
<FORM onSubmit="submitEvent()">
```

If you want to validate the information in the form and only allow it to be submitted if it meets your requirements, you will need to put a **return** statement that returns **false** in the event handler, as in the following example using the 'validate' function:

Code:
```
<FORM onSubmit="return validate(this)">
```

EVENT HANDLER: onResize

onResize = myJavaScriptCode

Event handler for **Window**

The **onResize** even handler is use to execute specified code whenever a user or script resizes a window or frame. This allows you to query the size and position of window elements, dynamically reset SRC properties etc. It uses the following properties of the **Event** object:

type - indicates the type of event.
target - indicates the target object to which the event was sent.
width, height - indicates the width or height of the window or frame

EVENT HANDLER: onUnload

onUnload = myJavaScriptCode

Event handler for **Window**

The **onUnload** event handler is used to run a function or JavaScript code whenever the user exits a document. The **onUnload** event handler is used within either the <BODY> or the <FRAMESET> tag, and uses the following properties of the **Event** object:

type - indicates the type of event
target - indicates the target object to which the event was sent.

The following example shows the **onUnload** event handler being used to execute the 'finishOff' function:

Code:
```
<BODY onUnload="finishOff()">
```

Compare to the **onload** event handler.

METHOD: RegExp::compile

---

object.compile(pattern[, flags])

This method compiles a regular expression object during execution of a script. It is used with a **RegExp** object created with the constructor function in order to compile it once only, this avoiding repeated compilation of a regular expression. This can be done once a regular expression has been got and you are sure that it will then remain constant throughout the rest of the script. The **compile** method can also be used to change and recompile a regular expression.

For example, suppose you created a regular expression consisting of the letters 'man' and then searched for a match in a string and replaced it with 'person' thus:

Code:
```
myRegExp = /man/
myString = "The Chairman of the Board"
newString = myString.replace(myRegExp, "person")
```

...the code would match the 'man' of 'Chairman' and replace it producing the word 'Chairperson'. If then you wanted to change the regular expression in order to replace either 'man' or 'woman' with 'person', you could do so using the **compile** method thus:

Code:
```
myRegExp.compile("(wo)?man")
newString = myString.replace(myRegExp, "person")
```

The **compile** method can also be used with the flags 'g' for a global match, 'i' for a case-insensitive match and 'gi' for a global, case-insensitive match. So, to expand on the above example, you could alter the regular expression 'MyRegExp' to search for all occurrences of the substrings 'man' and 'woman' and replace them with 'person' as follows:

Code:
```
myRegExp.compile("(wo)?man", "g")
newString = myString.replace(myRegExp, "person")
```

NOTE:

Calling the **compile** method alters the value of the **source**, **global** and **ignoreCase** properties of a regular expression.

METHOD: RegExp::exec

object.exec([str])

object([str])

This method executes a search for a match in a specified string, returning a result array. (If, however, you simply want to test whether or not there is a match, it is best to use the **test** method or the **String.search** method.) For example, the following blocks of code, one for the Internet Explorer browser and the other for Netscape, each execute a search of the string "the fisherman" for a match with the regular expression 'rexp'. If one is found (and in the case of the example it is), then an appropriate message is printed.

Code:

```
rexp = /[aeiou]/g
myString = "the fisherman"
if(rexp.exec(myString))
   match = rexp.exec(myString)
   document.write("Successfully matched " + match)
```

Code:

```
rexp = /[aeiou]/g
myString = "the fisherman"
if(rexp(myString))
   match = RegExp.lastMatch
   document.write("Successfully matched " + match)
```

Output:

```
Successfully matched e
```

NOTE:

The Netscape browser can call the **exec** method both directly (**object.exec([str])**) or indirectly (**object([str])**), whereas Microsoft Internet Explorer can only call it directly. If no string is declared then the value of **RegExp.input** is used. If the search fails, the **exec** method returns **null**.

METHOD: Layer::moveAbove

layer.moveAbove(layerName)

This method is used to move the layer above the one specified with the layerName argument. Performing this re-stacking does not alter the horizontal or vertical position of either layer. Note that after using this method, both layers share the same parent layer.

The following example creates two layers, aboveLayer and belowLayer, and calls the moveAbove method using the onMouseOver event handler of the belowLayer object.

Code:

```
<layer name=aboveLayer bgcolor="lightgreen" top=50 left=80 width=150 height=50>

aboveLayer

</layer>

<layer name=belowLayer above=aboveLayer bgcolor="lightblue" top=20 left=20 width=150
height=50 onMouseOver=moveAbove(aboveLayer)>

belowLayer

</layer>
```

METHOD: Layer::moveBelow

layer.moveBelow(layerName)

This method is used to move the layer below the one specified with the layerName argument. Performing this re-stacking does not alter the horizontal or vertical position of either layer. Note that after using this method, both layers share the same parent layer.

The following example creates two layers, aboveLayer and belowLayer, and calls the moveBelow method using the onMouseOver event handler of the aboveLayer object.

Code:

```
<layer name=aboveLayer bgcolor="lightgreen" top=50 left=80 width=150 height=50
onMouseOver=moveBelow(belowLayer)>

Mouse over me to reveal the layer below

</layer>

<layer name=belowLayer above=aboveLayer bgcolor="lightblue" top=20 left=20 width=150
height=50>

Hello from the layer below!

</layer>
```

METHOD: RegExp.test

object.test([str])

This method tests for a match of a regular expression in a string, returning **true** if successful, and **false** if not. The **test** method can be used with either a string literal or a string variable. For example, the following code tests for a match of the regular expression 'er' in the string "the fisherman", returning an appropriate message if a match is found (which in this case it is).

Code:
```
rexp = /er/
if(rexp.test("the fisherman"))
   document.write("It's true, I tell you.")
```

Output:
```
true
```

NOTE:

If no string is declared with this method, then the value of **RegExp.input** is used.

METHOD: Layer::load

---

layer.load("fileName", width)

This method is used to change the contents of a layer by loading a file containing HTML code into the layer. The width parameter alters the width in pixels at which the contents of the layer are wrapped.

The following example creates a layer and then changes its contents by loading an HTML file when the user moves the mouse over the layer.

Code:

```
<layer name=aboveLayer bgcolor="lightgreen" top=50 left=80 width=150 height=50
onMouseOver='load("myFile.html", 300)>

aboveLayer

</layer>
```

PROPERTY: RegExp::lastMatch

RegExp.lastMatch

This property is the last matched characters. As this property is static, you always use **RegExp.lastMatch**. For example, the following code would search through the string "the fisherman's tale" for a match with the regular expression of two consonants, one or more vowels and two consonants. In this case it will match the 'sherm' of 'fisherman'.

Code:
```
rexp = /([^aeiou\s]){2}([aeiou])+([^aeiou\s]){2}/
rexp("the fisherman's tale")
```

Output:
```
sherm
```

OBJECT: Anchor

String.anchor(nameAttribute)

An **Anchor** object is a place in a document that is the target of a hypertext link. There are two ways of creating it: by calling the **String.anchor** method or by using the HTML 'A' tag. Each of these tags that have a NAME attribute is placed by the JavaScript engine in an array in the **document.anchors** property. An **Anchor** object can then be accessed by indexing this array. The former uses code to produce the anchor using the **anchor** method along with the **document.write** or **document.writeln** method.

The following example creates an anchor called 'book_anchor' on the string 'INDEX OF BOOKS'

Code:
```
var mystring = "INDEX OF BOOKS"
document.write(mystring.anchor("book_anchor"))
```

Using the HTML 'A' tag, you can do exactly the same as the above as follows:

Code:
```
<A NAME="book_anchor">INDEX OF BOOKS</A>
```

The **Anchor** object inherits the **watch** and **unwatch** methods from **Object**, neither of which is supported by Microsoft JScript.

NOTE:

If the **Anchor** object is also a **Link** object, it'll have entries in both the **anchors** and **links** arrays.

OBJECT: Applet

An **Applet** object, created for every instance of the HTML <APPLET> tag in your document, allows the inclusion of a Java applet in a web page. These objects are then stored in an array in the **document.applets** property.

To enable an applet to access Javascript on your page, you must specify the <APPLET> tag's MAYSCRIPT attribute; failure to do this will cause an exception if the applet tries to access JavaScript. This allows a measure of security for each HTML page that contains the applet.

The following HTML code executes the myApp applet and sets the MAYSCRIPT attribute to allow it access to JavaScript. This will also automatically create an Applet object called "myApp" which will be added to the **document.applets** array and can be referenced as **document.applets[0]** (providing this is the first instance of the <APPLET> tag in your document) or **document.applets["myApp"]**.

Code:
```
<APPLET CODE="myApp.class" WIDTH=150 HEIGHT=80 NAME="myApp" MAYSCRIPT>
```

PROPERTIES

All public properties of a Java applet are inherited by the **Applet** object.

METHODS

All public methods of a Java applet are inherited by the **Applet** object.

Function: escape

escape(string)

The top-level function, **escape**, encodes the string that is contained in the **string** argument to make it portable. A string is considered portable if it can be transmitted across any network to any computer that supports ASCII characters.

To make a string portable, characters other than the following 69 ASCII characters must be encoded:

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
1234567890
@*-_+./

All other characters are converted either to their two digit (%xx) or four digit (%uxxxx) hexadecimal equivalent (refered to as the character's "hexadecimal escape sequence"). For example, a blank space will be represented by %20 and a semicolon by %3B. (Note that the hexadecimal numbers are: 0123456789ABCDEF).

Use the **unescape** function to decode an encoded sequence that was created using **escape**.

Code:
```
document.write(escape("Miss Piggy."))
```

Output:
```
Miss%20Piggy.
```

Code:
```
document.write(escape("!@#$%^&*()_+|"))
```

Output:
```
%21@%23%24%25%5E%26*%28%29_+%7C
```

Function: unescape

unescape(encodedstring)

The top-level function, **unescape**, decodes an encoded **string** argument that was created using the **escape** function.

The function searches for two and four digit hexidecimal escape sequences and replaces them in the string with their single character Latin-1 equivalent. For example, %3B signifies a semicolon.

Code:

```
document.write(unescape("Miss%20Piggy%20loves%20Kermit%21"))
```

Output:

```
Miss Piggy loves Kermit!
```

Function: eval

eval(codestring)

The top-level function, **eval**, evaluates and/or executes a string of JavaScript code that is contained in the **codestring** argument.

First, **eval** determines if the argument is a valid string and then parses the string looking for JavaScript code. If there are JavaScript statements in the code, they will be executed and **eval** will return the value of the last statement (if there is a value). If there is a JavaScript expression, it will be evaluated and its value will be returned.

Note that the **codestring** argument is optional. However, if there is no argument, **eval** returned, "undefined".

Code:
```
eval("fred=999; wilma=777; document.write(fred + wilma);");
```

Output:
```
1776
```

STATEMENT: export

export name1, name2, ..., nameN

export *

The **export** statement allows a signed script to provide properties, functions and objects to other signed or unsigned scripts. Usually a signed script can only pass information to another script signed by the same principals but this restriction can be overcome by the use of the **export** statement by the originating script and the accompanying **import** statement by the receiving script. The following code makes the 'wine' and 'beer' properties of the 'drinks' object available to any script wanting to import them (compare the **import** statement):

Code:

```
export drinks.beer, drinks.wine;
```

STATEMENT: import

import objectName.name1, objectName.name2, ..., objectName.nameN

import objectName.*

The **import** statement allows a script to import properties, functions and objects exported by a signed script. The following code imports the 'wine' and 'beer' properties of the object 'drinks' provided they have been made available by an exporting script (compare the **export** statement):

Code:

```
import drinks.beer, drinks.wine;
```

NOTE:

Any exported script must be loaded into a window, frame or layer before it can be imported and used.

STATEMENT: for...in

for (variable in object) {statements}

The **for...in** statement is used to iterate a declared variable over every property in a specified object. The code in the body of the **for ... in** loop is executed once for each property. The variable argument can be a named variable, an array element, or a property of the object.

This example simply displays the names of all the properties of the 'drink' object.

Code:
```
var i;
for(i in drink)
   document.write(i + "<BR>");
```

You can also have a specified variable iterate over the values of an object's properties by placing it between square brackets after the object name. Expanding on the previous example, the following code displays both the name of each property in the 'drink' object and its value:

Code:
```
var i;
for(i in drink)
   document.write(i + ":   " + drink[i] + "<BR>");
```

It is very easy to loop through an array. In this last example, starname is the name of an array element in an array called, starchart.

Code:
```
for(starname in starchart)
   document.write(starname + "<BR>");
```

OBJECT: Frame

A **Frame** object is created by using the HTML <FRAME> tag in a Window that contains the <FRAMESET> tag. A frame is an independent window within a parent window (in other words, you can display multiple frames, or windows, on a single screen). It has its own URL and is treated, with a few exceptions, as a **Window** object by JavaScript (this includes having all the same methods and properties of a **Window** object). For more information on **Frame** objects, go to the **Window** object page.

Primitive Value: NaN

NaN

In ECMAScript, **NaN** is classified as a primitive value.

NaN means "Not-a-Number". It is used to signify that a value is not a legal number.

Note that the primitive value, **Infinity**, is used to signify that a number, 1.5x10^{321} for example, has exceeded the defined range of legal values for a floating point number in JavaScript.

You can use the **isNan** function to test a value to see if it is a **NaN**.

Primitive Value: Infinity

Infinity

In ECMAScript, **Infinity** is classified as a primitive value.

Infinity is a numeric value that represents positive infinity. It is displayed, or printed out, when a very large positive number exceeds the upper limit of the floating point numbers type which is 1.7976931348623157E+10^{308}.

-Infinity is a numeric value that represents negative infinity. It is displayed, or printed out, when a very large negative number exceeds the lower limit of the floating point numbers type which is -1.7976931348623157E+10^{308}.

Also see number.NEGATIVE_INFINITY and number.POSITIVE_INFINITY.

Code:
```
BigPosNum = 1.5E+339 * 2.4E+317
document.write("BigPosNum = " + BigPosNum)
```

Output:
```
BigNum = Infinity
```

Code:
```
BigNegNum = -1.5E+333
document.write("BigNegNum = " + BigNegNum)
```

Output:
```
BigNegNum = -Infinity
```

NOTE:

In JavaScript, all numbers, including integers, are treated as floating point numbers.

FUNCTION: isNaN

isNaN(testvalue)

The **isNaN** function is used to determine if the argument, **testvalue**, is a **NaN**.

A **NaN**, which means "Not-a-Number", is classified as a primitive value by the ECMA-262 standard and indicates that the specified value is not a legal number. The function returns **true** if the argument is not a number and **false** if the argument is a number.

The classic example of a **NaN** is zero divided by zero, 0/0.

Code:
```
document.write(isNaN("Ima String"))
document.write(isNaN(0/0))
document.write(isNaN("348"))
document.write(isNaN(348))
```

Output:
```
true
true
false
false
```

Function: number

number(object)

The top-level function, **number**, converts the **object** argument to a string representing the object's value. If the value cannot be represented by a legitimate number, the "Not-a-Number" value, **NaN** is returned.

The **object** argument must be a JavaScript object. If no argument is provided, **number** returns zero, 0.

In this example, a new **Boolean** object is created with a string argument of "true". The **number** function returns, 1, which is the number equivalence of the value of the object.

Code:

```
boo = new Boolean("true")
document.write(Number(boo))
```

Output:

```
1
```

Function: string

string(object)

The top-level function, **string**, converts the **object** argument to a string representing the object's value.

The **object** argument must be a JavaScript object. If no argument is provided, **string** returns the empty string, "".

In this example, a new **Boolean** object is created with an argument of 0. The **String** function returns, "false", which is the string equivalence of the value of the object.

Code:
```
boo = new Boolean(0)
document.write(String(boo))
```

Output:
```
false
```

STATEMENT: break

break [label]

The **break** statement can be used to terminate a current loop, switch or label statement and pass control to the statement immediately following it.

The following example starts a loop printing the numbers from 1 to 10, buts exits when it reaches 7 with an appropriate message:

Code:
```
var i = 0
while (i < 10)
{
   document.write(i);
   if (i==7)
   {
      document.write("the counter has reached " + i);
      break;
   }
   i++;
}
```

The **break** statement can also be used with a label as in the following example of two counts, one nested within the other, which will be ended if the inner counter variable is equal to the variable 'x':

Code:
```
outer_loop:
for(i=0; i<3; i++)
{
   document.write("<BR>" + "outer " + i + ":   ");
   for(j=0; j<5; j++)
   {
      document.write("inner " + j + " ");
      if(j==x)
         break outer_loop;
   }
}
```

While the **break** statement on its own can only be used to exit a loop, the optional label can be added to **break** to exit any kind of statement. This next example tests for an even number and, whenever it finds one, displays it, unless that number is 12:

Code:
```
even_number:
if(i%2==0)
{
   if(i==12)
      break even_number;
   document.write(i);
}
```

Function: parseFloat

parseFloat(string)

The top-level function, **parseFloat**, finds the first number in a string.

The function determines if the first character in the **string** argument is a number, parses the string from left to right until it reaches the end of the number, discards any characters that occur after the end of the number, and finally returns the number as a number (not as a string).

Only the first number in the string is returned, regardless of how many other numbers occur in the string.

If the first character in the string is not a number, the function returns the Not-a-Number value **NaN**.

Code:
```
document.write("<BR>" + parseFloat("50"))
document.write("<BR>" + parseFloat("50.12345"))
document.write("<BR>" + parseFloat("32.00000000"))
document.write("<BR>" + parseFloat("71.348  92.218  95.405"))
document.write("<BR>" + parseFloat("37 aardvarks"))
document.write("<BR>" + parseFloat("Awarded the best wine of 1999"))
```

Output:
```
50
50.12345
32.00000000
71.348
37
NaN
```

You can use the **isNaN** function to see if the returned value is a **NaN**.

Code:
```
cost = "$99.88"
CheckNum = parseFloat(cost)
if(isNaN(CheckNum))
   {
     document.write("<BR>Sorry, CheckNum is a NaN")
     document.write("<BR>Left-most character = " + cost.substring(0,1))
   }
```

Output:
```
Sorry, CheckNum is a NaN
Left-most character = $
```

Function: parseInt

parseInt(string, radix)

The top-level function, **parseInt**, find the first integer in a string.

There are two arguments. The mandatory **string** argument is the string which is presumed to contain an integer. The optional **radix** argument specifies the base of the integer and can range from 2 to 36. For example, 16 is the hexidecimal base (0123456789ABCDEF).

If no **radix** argument is provided or if it is assigned a value of 0, the function tries to determine the base. If the string starts with a 1-9, it will be parsed as base 10. If the string starts with 0x or 0X it will be parsed as a hexidecimal number. If the string starts with a 0 it will be parsed as an octal number. (Note that just because a number starts with a zero, it does not mean that it is really octal.)

After determining if the first character in the **string** argument is a number, the **parseInt** function parses the string from left to right until the end of the number or a decimal point is encountered, then it discards any characters that occur after the end of the number (including a decimal point and all numbers after the decimal point), and finally it returns the number as an integer (not as a string).

Only the first integer in the string is returned, regardless of how many other numbers occur in the string. This function does not return decimal points nor numbers to the right of a decimal.

If the first non-whitespace character is not numeric, the function returns the Not-a-Number value **NaN**.

Code:
```
document.write("<BR>" + parseInt("50"))
document.write("<BR>" + parseInt("50.12345"))
document.write("<BR>" + parseInt("32.00000000"))
document.write("<BR>" + parseInt("71.348  92.218  95.405"))
document.write("<BR>" + parseInt("        37 aardvarks"))
document.write("<BR>" + parseInt("Awarded the best wine of 1992"))
```

Output:
```
50
50
32
71
37
NaN
```

You can use the optional **radix** argument to convert a binary number string to the decimal (base 10) equivalent.

Code:
```
document.write(parseInt("110", 2))
```

Output:
```
6
```

You can use the optional **radix** argument to convert a hexidecimal number string to the decimal (base 10) equivalent.

Code:
```
document.write(parseInt("0xD9", 16))
```

Output:
```
217
```

You can use the **isNaN** function to see if the returned value is a **NaN**.

Code:
```
pet = "Rachel has 312 baby aardvarks"
CheckNum = parseInt(pet)
if(isNaN(CheckNum))
   {
     document.write("<BR>Sorry, CheckNum is a NaN")
     document.write("<BR>Left-most character = " + pet.substring(0,1))
   }
```

Output:
```
Sorry, CheckNum is a NaN
Left-most character = R
```

STATEMENT: switch

switch

The **switch** statement tests an expression against a number of **case** options and executes the statements associated with the first one to match. If no match is found, the program looks for a set of default statements to execute, and if these aren't found either, it carries on with the statement immediately following switch. An optional **break** statement with each case ensures that once a set of statements has been executed, the program exits switch. If **break** were omitted, the statements associated with the following case would also be executed:

Code:
```
switch (i)
{
  case "Chicago" :
    document.write ("Flights to Chicago: Saturdays.");
    break;
  case "London" :
    document.write ("Flights to London: Fridays.");
    break;
  case "New York" :
    document.write ("Flights to New York: Fridays.");
    break;
  case "San Francisco" :
    document.write ("Flights to San Francisco: Wednesdays.");
    break;
  default :
    document.write ("Sorry, there are no flights to " + i + ".<BR>");
}
document.write("Thank you for enquiring with Northern Airlines.<BR>");
```

Function: taint

taint

The **taint** function is deprecated.

The Data-Tainting Security Model, which used the **taint** and **untaint** functions, was tried on an experimental basis in Navigator 3. The idea behind the concept was to prevent private data was being accessed on the Web. It proved unsuccessful and no code examples will be provided.

STATEMENT: throw

throw (exception)

The **throw** statement allows the programmer to create an exception. This **exception** can be a string, integer, Boolean or an object. Coupling the **throw** statement with the **try...catch** statement, the programmer can control program flow and generate accurate error messages.

The following example determines the value of variable(z), and generates an error accordingly. This error is then caught by the **catch** argument and the proper error message is then displayed.

Code:

```
try {
   if(z == 1)
      throw "Error 1"
   else if(z == 2)
      throw "Error 2"
}
catch(er) {
   if(er == "Error 1")
      alert("Error 1 Please contact system Administrator")
   if(er == "Error 2")
      alert("Error 2 Please Reload the page")
}
```

STATEMENT: try...catch

try {statements1} [catch (exception){statements2}]

The **try...catch** statement is used to test a block of code for errors. The **try** block contains the code to be run, while the **catch** block contains the code to execute if there is an error. The exception argument is a variable in which to store the error, in this case it is the variable er.

The following example determines the value of variable(z), and generates an error accordingly. This error is then caught by the **catch** argument and the proper error message is then displayed.

Code:
```
try {
   if(z == 1)
      throw "Error 1"
   else if(z == 2)
      throw "Error 2"
}
catch(er) {
   if(er == "Error 1")
      alert("Error 1 Please contact system Administrator")
   if(er == "Error 2")
      alert("Error 2 Please Reload the page")
}
```

STATEMENT: // /*...*/

// comment text

/* multiple line comment text */

Comments are notes by the author explaining what the script does, and are ignored by the interpreter. A comment consisting of a single line is preceded by a double slash (//):

Code:

```
// This is a single-line comment.
```

...and a multiple line comment is preceded by a /* and followed by a */:

Code:

```
/* This is a multiple line comment. It can contain whatever letters and characters you like and
span as many lines as you like. */
```

STATEMENT: continue

continue **[label]**

The **continue** statement is used to restart a **while**, **do...while**, **for** or **label** statement. In a **while** loop it jumps back to the condition.

In the following example the code produces the numbers 1 thru 10 but skips the number 7:

Code:
```
var i = 0
while (i < 10)
{
  i++;
  if (i==7)
    continue;
  document.write(i + ".<BR>");
}
```

...while in a **for** loop it jumps back to the update expression, as in the next example which lists all the elements of the array 'drink' except for when the array index equals 2 (Note: since the array indexing starts at zero, index 2 is the 3rd element in the array):

Code:
```
for(i=0; i<4; i++)
{
  if (i==2)
    continue;
  document.write(drink[i]);
}
```

It can also be used with a label as in the next example which displays an outer and inner count, but limits the inner count to 3 more than the outer:

Code:
```
count_loop:
for(i=0; i<3; i++)
{
  document.write("<BR>" + "outer " + i + ":   ");
  for(j=0; j<10; j++)
  {
    document.write("inner " + j + " ");
    if(j==i+3)
      continue count_loop;
  }
}
```

STATEMENT: do...while

do statements while (condition);

The **do...while** statement executes one or more statements at least once, checking that a certain condition is met each time before repeating. If that condition is not met, then control moves to the statement immediately after the loop. The following example counts up in twos for as long as the number is less than 20:

```
var i = 0;
do
{
   document.write(i + ".<BR>");
   i+=2;
}
while(i<20);
```

FUNCTION: isFinite

isFinite(testnumber)

The top-level function, **isFinite**, is used to determine if the argument, **testnumber**, is a finite and legal number. This function returns **true** for a finite number and otherwise returns **false**.

Code:
```
document.write(isFinite(2.2345))
```

Output:
```
true
```

Code:
```
document.write(isFinite(2.5E+345))
```

Output:
```
false
```

Code:
```
document.write(isFinite("Ima string"))
```

Output:
```
false
```

STATEMENT: label

label: statements;

A label can be used to identify a statement allowing you to refer to it elsewhere in a program. It can be used with the **break** or **continue** statements (examples of which can be seen in the relevent pages) to modify the execution of a loop. It can also be used with other statements as in the following example which tests for an even number, and displays it, unless it's a 12:

Code:
```
even_number:
if(i%2==0)
{
   if(i==12)
      break even_number;
   document.write(i);
}
```

Primitive Value: Undefined

Undefined

In ECMAScript, **Undefined** is classified as a primitive value. Your ability to use **Undefined** will be extremely dependent upon the type and version of your browser.

There are two definitions for **Undefined**. It can refer to a variable that has never been declared. Or it can refer to a variable that has been declared, but has not been assigned a value. The ECMA-262 standard uses the second version to define **Undefined**.

Undefined is also a type. You can use the **typeof** operator to determine the type of a variable and it will return a type of "undefined" for an **Undefined** variable.

For this example, the value NotThere has not been declared. (Note that it is optional to enclose the argument for the **typeof** operator inside a pair of parenthesis.)

Code:
```
document.write("NotThere is of type = " + typeof NotThere)
```

Output:
```
NotThere is of type = undefined
```

In this example, the value IsThere has been declared, but it is not assigned a value.

Code:
```
var IsThere;
document.write("IsThere is of type = " + typeof IsThere)
```

Output:
```
IsThere is of type = undefined
```

NOTE:

In Internet Explorer, if you attempt to utilize an undefined variable, you will get a runtime error message b

Function: untaint

---

untaint

The **untaint** function is deprecated.

The Data-Tainting Security Model, which used the **taint** and **untaint** functions, was tried on an experimental basis in Navigator 3. The idea behind the concept was to prevent private data was being accessed on the Web. It proved unsuccessful and no code examples will be provided.

STATEMENT: var

var

The **var** statement is used to declare a variable, and outside of a function its use is optional. While a variable can be declared simply by assigning it a value, it is good practice to use **var** as there are two cases in functions where it is necessary:

If a global variable of the same name exists.

If recursive or multiple functions use variables of the same name.

Code:

```
var i;
```

You can also declare more than one variable and, optionally, assign values at the same time:

Code:

```
var i = 0, x, max = 57;
```

STATEMENT: while

while

The **while** statement creates a loop consisting of a block of statements that is executed if the expression evaluated is true.

The following example simply counts 1 thru 10 by incrementing a counter by 1 each time for as long as the counter is less than 11:

Code:
```
var i = 0;
while(i<11)
{
   document.write(i + "<BR>");
   i++;
}
```

FUNCTIONS

STATEMENTS

OPERATORS

Arithmetic

=
++
-
--
/
%

Assignment

=

Backslash Escaped Characters

\'
\"
\\
\b
\f
\n
\r
\t

Bitwise

&
|
^
~
<<
>>
>>>

Comparison

==
!=
===
!==
>
>=
<
<=

Logical

&&
||
!

Special

?:
'
delete
new
this
typeof
void

String

+

VALUES

Infinity
NaN
Undefined

OBJECTS

Anchor
Applet
Area
Array
Boolean
Button
Checkbox
Date
Document
Event
FileUpload
Form
Frame
Function
Hidden
History
Image
Layer

Link
Location
Math
Navigator
Number
Object
Option
Password
Radio
RegExp
Reset
Screen
Select
String
Submit
Text
Textarea
Window