Core JavaScript Reference

This book is a reference manual for the core JavaScript language for version 1.5. JavaScript is Netscape's cross-platform, object-based scripting language. Core JavaScript can be extended for a variety of purposes by supplementing it with additional objects.

Index **Next**

Last Updated **September 28, 2000**

About this Book

JavaScript is Netscape's cross-platform, object-based scripting language. This book is a reference manual for the core JavaScript language.

This preface contains the following sections:

New Features in this Release

JavaScript version 1.5 provides the following new features and enhancements:

- **Runtime errors.** Runtime errors are now reported as exceptions.
- **Number formatting enhancements.** Number formatting has been enhanced to include Number.prototype.toExponential, Number.protoytpe.toFixed and Number.prototype.toPrecision methods. See page 127, page 128, and page 129.
- **Regular expression enhancements:**
  - Greedy quantifiers - +, *, ? and {} - can now be followed by a ? to force them to be non-greedy. See the entry for ? on page 148.

    - Non-capturing parentheses, (?:x) can be used instead of capturing parentheses(x). When non-capturing parentheses are used, matched subexpressions are not available as back-references. See the entry for (?:x) on page 148.

    - Positive and negative lookahead assertions are supported. Both assert a match depending on what follows the string being matched. See the entries for (?=) and (?!) on page 148.

    - The m flag has been added to specify that the regular expression should match over multiple lines. See page 146.

- **Conditional function declarations.** Functions can now be declared inside an **if** clause. See page 221.

- **Function expressions.** Functions can now be declared inside an expression. See page 254.

- **Multiple catch clauses.** Multiple catch clauses in a try...catch statement are supported. See page 231.

- **Constants.** Readonly, named constants are supported. This feature is available only in the C implementation of JavaScript. See page 215.

- **Getters and Setters.** JavaScript writers can now add getters and setters to their objects. This feature is available only in the C implementation of JavaScript. See Defining Getters and Setters in Chapter 7 of the *Core JavaScript Guide* for information about this feature.

What You Should Already Know

This book assumes you have the following basic background:

-
- A general understanding of the Internet and the World Wide Web (WWW).

- Good working knowledge of HyperText Markup Language (HTML).

Some programming experience with a language such as C or Visual Basic is useful, but not required.

JavaScript Versions

Each version of Navigator supports a different version of JavaScript. To help you write scripts that are compatible with multiple versions of Navigator, this manual lists the JavaScript version in which each feature was implemented.

The following table lists the JavaScript version supported by different Navigator versions. Versions of Navigator prior to 2.0 do not support JavaScript.

Table 1 JavaScript and Navigator versions

JavaScript version	Navigator version
JavaScript 1.0	Navigator 2.0
JavaScript 1.1	Navigator 3.0
JavaScript 1.2	Navigator 4.0-4.05
JavaScript 1.3	Navigator 4.06-4.7x
JavaScript 1.4	-
JavaScript 1.5	Navigator 6.0 Mozilla (open source browser)

Each version of the Netscape Enterprise Server also supports a different version of JavaScript. To help you write scripts that are compatible with multiple versions of the Enterprise Server, this manual uses an abbreviation to indicate the server version in which each feature was implemented.

Table 2 JavaScript and Netscape Enterprise Server versions

Abbreviation	Enterpriser Server version
NES 2.0	Netscape Enterprise Server 2.0
NES 3.0	Netscape Enterprise Server 3.0

Where to Find JavaScript Information

The core JavaScript documentation includes the following books:

- The *Core JavaScript Guide* provides information about the core JavaScript language and its objects.
- The *Core JavaScript Reference* (this book) provides reference material for the core JavaScript language.

If you are new to JavaScript, start with the *Core JavaScript Guide*. Once you have a firm grasp of the fundamentals, you can use the *Core JavaScript Reference* to get more details on individual objects and statements.

Document Conventions

JavaScript applications run on many operating systems; the information in this book applies to all versions. File and directory paths are given in Windows format (with backslashes separating directory names). For Unix versions, the directory paths are the same, except that you use slashes instead of backslashes to separate directories.

This book uses uniform resource locators (URLs) of the following form:

http://*server.domain/path/file*.htmll

In these URLs, *server* represents the name of the server on which you run your application, such as research1 or www; *domain* represents your Internet domain name,

such as netscape.com or uiuc.edu; *path* represents the directory structure on the server; and *file*.htmll represents an individual file name. In general, items in italics in URLs are placeholders and items in normal monospace font are literals. If your server has Secure Sockets Layer (SSL) enabled, you would use https instead of http in the URL.

This book uses the following font conventions:

-
- The monospace font is used for sample code and code listings, API and language elements (such as method names and property names), file names, path names, directory names, HTML tags, and any text that must be typed on the screen. (*Monospace italic font* is used for placeholders embedded in code.)

- *Italic type* is used for book titles, emphasis, variables and placeholders, and words used in the literal sense.

- **Boldface type** is used for glossary terms.

Previous **Contents** **Index** **Next**

Last Updated **September 28, 2000**

Part 1 Object Reference

Last Updated **September 28, 2000**

Chapter 1 Chapter 1 Objects, Methods, and Properties

This chapter documents all the JavaScript objects, along with their methods and properties. It is an alphabetical reference for the main features of JavaScript.

The reference is organized as follows:

-
- Full entries for each object appear in alphabetical order; properties and functions not associated with any object appear in Chapter 2, "Top-Level Properties and Functions."
  Each entry provides a complete description for an object. Tables included in the description of each object summarize the object's methods and properties.
- Full entries for an object's methods and properties appear in alphabetical order after the object's entry.
  These entries provide a complete description for each method or property, and include cross-references to related features in the documentation.

Last Updated **September 28, 2000**

Array

Lets you work with arrays.

Core object	
Implemented in	JavaScript 1.1, NES 2.0 JavaScript 1.3: added toSource method; changed length property; changed push method
ECMA version	ECMA-262

Created by
The Array object constructor:

new Array(*arrayLength*)
new Array(*element0*, *element1*, ..., *elementN*)

An array literal:

[*element0*, *element1*, ..., *elementN*]

JavaScript 1.2 when you specify LANGUAGE="JavaScript1.2" in the <SCRIPT> tag:

new Array(*element0*, *element1*, ..., *elementN*)

JavaScript 1.2 when you do not specify LANGUAGE="JavaScript1.2" in the

<SCRIPT> tag:

new Array([*arrayLength*])
new Array([*element0*[, *element1*[, ..., *elementN*]]])

JavaScript 1.1:

new Array([*arrayLength*])
new Array([*element0*[, *element1*[, ..., *elementN*]]])

Parameters

arrayLength	The initial length of the array. You can access this value using the length property. If the value specified is not a number, an array of length 1 is created, with the first element having the specified value. The maximum length allowed for an array is 4,294,967,295.
element*N*	A list of values for the array's elements. When this form is specified, the array is initialized with the specified values as its elements, and the array's length property is set to the number of arguments.

Description
An array is an ordered set of values associated with a single variable name.

The following example creates an Array object with an array literal; the coffees array contains three elements and a length of three:

coffees = ["French Roast", "Columbian", "Kona"]

You can construct a *dense* array of two or more elements starting with index 0 if you define initial values for all elements. A dense array is one in which each element has a value. The following code creates a dense array with three elements:

myArray = new Array("Hello", myVar, 3.14159)

Indexing an array. You index an array by its ordinal number. For example, assume you define the following array:

myArray = new Array("Wind","Rain","Fire")

You then refer to the first element of the array as myArray[0] and the second element of the array as myArray[1].

Specifying a single parameter. When you specify a single numeric parameter with the Array constructor, you specify the initial length of the array. The following code creates an array of five elements:

billingMethod = new Array(5)

The behavior of the Array constructor depends on whether the single parameter is a number.

-
- If the value specified is a number, the constructor converts the number to an unsigned, 32-bit integer and generates an array with the length property (size of the array) set to the integer. The array initially contains no elements, even though it might have a non-zero length.

- If the value specified is not a number, an array of length 1 is created, with the first element having the specified value.

The following code creates an array of length 25, then assigns values to the first three elements:

```
musicTypes = new Array(25)
musicTypes[0] = "R&B"
musicTypes[1] = "Blues"
musicTypes[2] = "Jazz"
```

Increasing the array length indirectly. An array's length increases if you assign a value to an element higher than the current length of the array. The following code creates an array of length 0, then assigns a value to element 99. This changes the length of the array to 100.

```
colors = new Array()
colors[99] = "midnightblue"
```

Creating an array using the result of a match. The result of a match between a

regular expression and a string can create an array. This array has properties and elements that provide information about the match. An array is the return value of RegExp.exec, String.match, and String.replace. To help explain these properties and elements, look at the following example and then refer to the table below:

```
<SCRIPT LANGUAGE="JavaScript1.2">
//Match one d followed by one or more b's followed by one d
//Remember matched b's and the following d
//Ignore case

myRe=/d(b+)(d)/i;
myArray = myRe.exec("cdbBdbsbz");

</SCRIPT>
```

The properties and elements returned from this match are as follows:

Property/Element	Description	Example
input	A read-only property that reflects the original string against which the regular expression was matched.	cdbBdbsbz
index	A read-only property that is the zero-based index of the match in the string.	1
[0]	A read-only element that specifies the last matched characters.	dbBd

[1], ...[n]	Read-only elements that specify the parenthesized substring matches, if included in the regular expression. The number of possible parenthesized substrings is unlimited.	[1]=bB [2]=d

Backward Compatibility

JavaScript 1.2. When you specify a single parameter with the Array constructor, the behavior depends on whether you specify LANGUAGE="JavaScript1.2" in the <SCRIPT> tag:

-
- If you specify LANGUAGE="JavaScript1.2" in the <SCRIPT> tag, a single-element array is returned. For example, new Array(5) creates a one-element array with the first element being 5. A constructor with a single parameter acts in the same way as a multiple parameter constructor. You cannot specify the length property of an Array using a constructor with one parameter.

- If you do not specify LANGUAGE="JavaScript1.2" in the <SCRIPT> tag, you specify the initial length of the array as with other JavaScript versions.

JavaScript 1.1 and earlier. When you specify a single parameter with the Array constructor, you specify the initial length of the array. The following code creates an array of five elements:

billingMethod = new Array(5)

JavaScript 1.0. You must index an array by its ordinal number; for example myArray[0].

Property Summary

Property	Description

constructor	Specifies the function that creates an object's prototype.
index	For an array created by a regular expression match, the zero-based index of the match in the string.
input	For an array created by a regular expression match, reflects the original string against which the regular expression was matched.
length	Reflects the number of elements in an array.
prototype	Allows the addition of properties to all objects.

Method Summary

Method	Description
concat	Joins two arrays and returns a new array.

join	Joins all elements of an array into a string.
pop	Removes the last element from an array and returns that element.
push	Adds one or more elements to the end of an array and returns the new length of the array.
reverse	Transposes the elements of an array: the first array element becomes the last and the last becomes the first.
shift	Removes the first element from an array and returns that element.
slice	Extracts a section of an array and returns a new array.
splice	Adds and/or removes elements from an array.
sort	Sorts the elements of an array.

toSource	Returns an array literal representing the specified array; you can use this value to create a new array. Overrides the Object.toSource method.
toString	Returns a string representing the array and its elements. Overrides the Object.toString method.
unshift	Adds one or more elements to the front of an array and returns the new length of the array.
valueOf	Returns the primitive value of the array. Overrides the Object.valueOf method.

In addition, this object inherits the watch and unwatch methods from Object.

Examples

Example 1. The following example creates an array, msgArray, with a length of 0, then assigns values to msgArray[0] and msgArray[99], changing the length of the array to 100.

```
msgArray = new Array()
msgArray[0] = "Hello"
msgArray[99] = "world"
// The following statement is true,
// because defined msgArray[99] element.
if (msgArray.length == 100)
   myVar="The length is 100."
```

Example 2: Two-dimensional array. The following code creates a two-dimensional array and assigns the results to myVar.

```
myVar="Multidimensional array test; "
a = new Array(4)
for (i=0; i < 4; i++) {
   a[i] = new Array(4)
```

```
  for (j=0; j < 4; j++) {
    a[i][j] = "["+i+","+j+"]"
  }
}
for (i=0; i < 4; i++) {
  str = "Row "+i+":"
  for (j=0; j < 4; j++) {
    str += a[i][j]
  }
  myVar += str +"; "
}
```

This example assigns the following string to myVar (line breaks are used here for readability):

Multidimensional array test;
Row 0:[0,0][0,1][0,2][0,3];
Row 1:[1,0][1,1][1,2][1,3];
Row 2:[2,0][2,1][2,2][2,3];
Row 3:[3,0][3,1][3,2][3,3];

concat

Joins two arrays and returns a new array.

Method of	Array
Implemented in	JavaScript 1.2, NES 3.0
ECMA version	ECMA-262

Syntax
concat(*arrayName2, arrayName3, ..., arrayNameN*)

Parameters

arrayName2... arrayName*N*	Arrays to concatenate to this array.

Description

concat does not alter the original arrays, but returns a "one level deep" copy that contains copies of the same elements combined from the original arrays. Elements of the original arrays are copied into the new array as follows:

-
- Object references (and not the actual object): concat copies object references into the new array. Both the original and new array refer to the same object. If a referenced object changes, the changes are visible to both the new and original arrays.

- Strings and numbers (not String and Number objects): concat copies strings and numbers into the new array. Changes to the string or number in one array does not affect the other arrays.

If a new element is added to either array, the other array is not affected.

The following code concatenates two arrays:

```
alpha=new Array("a","b","c")
numeric=new Array(1,2,3)
alphaNumeric=alpha.concat(numeric) // creates array ["a","b","c",1,2,3]
```

The following code concatenates three arrays:

```
num1=[1,2,3]
num2=[4,5,6]
num3=[7,8,9]
nums=num1.concat(num2,num3) // creates array [1,2,3,4,5,6,7,8,9]
```

constructor

Specifies the function that creates an object's prototype. Note that the value of this property is a reference to the function itself, not a string containing the function's name.

Property of	Array
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description
See Object.constructor.

index

For an array created by a regular expression match, the zero-based index of the match in the string.

Property of	Array
Static	
Implemented in	JavaScript 1.2, NES 3.0

input

For an array created by a regular expression match, reflects the original string against which the regular expression was matched.

Property of	Array
Static	
Implemented in	JavaScript 1.2, NES 3.0

join

Joins all elements of an array into a string.

Method of	Array
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Syntax
join(*separator*)

Parameters

separator	Specifies a string to separate each element of the array. The separator is converted to a string if necessary. If omitted, the array elements are separated with a comma.

Description
The string conversions of all array elements are joined into one string.

Examples
The following example creates an array, a, with three elements, then joins the array three times: using the default separator, then a comma and a space, and then a plus.

```
a = new Array("Wind","Rain","Fire")
myVar1=a.join()      // assigns "Wind,Rain,Fire" to myVar1
myVar2=a.join(", ")  // assigns "Wind, Rain, Fire" to myVar1
myVar3=a.join(" + ") // assigns "Wind + Rain + Fire" to myVar1
```

See also
Array.reverse

length

An unsigned, 32-bit integer that specifies the number of elements in an array.

Property of	Array

Implemented in	JavaScript 1.1, NES 2.0 JavaScript 1.3: length is an unsigned, 32-bit integer with a value less than 2^{32}.
ECMA version	ECMA-262

Description

The value of the length property is an integer with a positive sign and a value less than 2 to the 32 power (2^{32}).

You can set the length property to truncate an array at any time. When you extend an array by changing its length property, the number of actual elements does not increase; for example, if you set length to 3 when it is currently 2, the array still contains only 2 elements.

Examples

In the following example, the getChoice function uses the length property to iterate over every element in the musicType array. musicType is a select element on the musicForm form.

```
function getChoice() {
  for (var i = 0; i < document.musicForm.musicType.length; i++) {
    if (document.musicForm.musicType.options[i].selected == true) {
      return document.musicForm.musicType.options[i].text
    }
  }
}
```

The following example shortens the array statesUS to a length of 50 if the current length is greater than 50.

```
if (statesUS.length > 50) {
  statesUS.length=50
}
```

pop

Removes the last element from an array and returns that element. This method changes the length of the array.

Method of	Array
Implemented in	JavaScript 1.2, NES 3.0
ECMA version	ECMA-262 Edition 3

Syntax
pop()

Parameters
None.

Example
The following code creates the myFish array containing four elements, then removes its last element.

```
myFish = ["angel", "clown", "mandarin", "surgeon"];
popped = myFish.pop();
```

See also
push, shift, unshift

prototype

Represents the prototype for this class. You can use the prototype to add properties or methods to all instances of a class. For information on prototypes, see Function.prototype.

Property of	Array
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

push

Adds one or more elements to the end of an array and returns the new length of the array. This method changes the length of the array.

Method of	Array
Implemented in	JavaScript 1.2, NES 3.0 JavaScript 1.3: push returns the new length of the array rather than the last element added to the array.
ECMA version	ECMA-262 Edition 3

Syntax
push(*element1*, ..., *elementN*)

Parameters

element1, ..., element*N*	The elements to add to the end of the array.

Description
The behavior of the push method is analogous to the push function in Perl 4. Note that this behavior is different in Perl 5.

Backward Compatibility

JavaScript 1.2. The push method returns the last element added to an array.

Example
The following code creates the myFish array containing two elements, then adds two elements to it. After the code executes, pushed contains 4. (In JavaScript 1.2, pushed contains "lion" after the code executes.)

```
myFish = ["angel", "clown"];
pushed = myFish.push("drum", "lion");
```

See also
pop, shift, unshift

reverse

Transposes the elements of an array: the first array element becomes the last and the last becomes the first.

Method of	Array
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Syntax
reverse()

Parameters
None

Description
The reverse method transposes the elements of the calling array object.

Examples
The following example creates an array myArray, containing three elements, then reverses the array.

```
myArray = new Array("one", "two", "three")
myArray.reverse()
```

This code changes myArray so that:

-
- myArray[0] is "three"
- myArray[1] is "two"
- myArray[2] is "one"

See also
Array.join, Array.sort

shift

Removes the first element from an array and returns that element. This method changes the length of the array.

Method of	Array
Implemented in	JavaScript 1.2, NES 3.0
ECMA version	ECMA-262 Edition 3

Syntax
shift()

Parameters
None.

Example
The following code displays the myFish array before and after removing its first element. It also displays the removed element:

```
myFish = ["angel", "clown", "mandarin", "surgeon"];
document.writeln("myFish before: " + myFish);
shifted = myFish.shift();
document.writeln("myFish after: " + myFish);
document.writeln("Removed this element: " + shifted);
```

This example displays the following:

```
myFish before: ["angel", "clown", "mandarin", "surgeon"]
myFish after: ["clown", "mandarin", "surgeon"]
Removed this element: angel
```

See also
pop, push, unshift

slice

Extracts a section of an array and returns a new array.

Method of	Array
Implemented in	JavaScript 1.2, NES 3.0
ECMA version	ECMA-262 Edition 3

Syntax
slice(*begin*[,*end*])

Parameters

begin	Zero-based index at which to begin extraction.

end	Zero-based index at which to end extraction: • slice extracts up to but not including end. slice(1,4) extracts the second element through the fourth element (elements indexed 1, 2, and 3). • As a negative index, end indicates an offset from the end of the sequence. slice(2,-1) extracts the third element through the second to last element in the sequence. • If end is omitted, slice extracts to the end of the sequence.

Description

slice does not alter the original array, but returns a new "one level deep" copy that contains copies of the elements sliced from the original array. Elements of the original array are copied into the new array as follows:

- For object references (and not the actual object), slice copies object references into the new array. Both the original and new array refer to the same object. If a referenced object changes, the changes are visible to both the new and original arrays.
- For strings and numbers (not String and Number objects), slice copies strings and numbers into the new array. Changes to the string or number in one array does not affect the other array.

If a new element is added to either array, the other array is not affected.

Example

In the following example, slice creates a new array, newCar, from myCar. Both include a reference to the object myHonda. When the color of myHonda is changed to purple, both arrays reflect the change.

```
<SCRIPT LANGUAGE="JavaScript1.2">

//Using slice, create newCar from myCar.
myHonda = {color:"red",wheels:4,engine:{cylinders:4,size:2.2}}
```

```
myCar = [myHonda, 2, "cherry condition", "purchased 1997"]
newCar = myCar.slice(0,2)

//Write the values of myCar, newCar, and the color of myHonda
// referenced from both arrays.
document.write("myCar = " + myCar + "<BR>")
document.write("newCar = " + newCar + "<BR>")
document.write("myCar[0].color = " + myCar[0].color + "<BR>")
document.write("newCar[0].color = " + newCar[0].color + "<BR><BR>")

//Change the color of myHonda.
myHonda.color = "purple"
document.write("The new color of my Honda is " + myHonda.color + "<BR><BR>")

//Write the color of myHonda referenced from both arrays.
document.write("myCar[0].color = " + myCar[0].color + "<BR>")
document.write("newCar[0].color = " + newCar[0].color + "<BR>")

</SCRIPT>
```

This script writes:

```
myCar = [{color:"red", wheels:4, engine:{cylinders:4, size:2.2}}, 2,
   "cherry condition", "purchased 1997"]
newCar = [{color:"red", wheels:4, engine:{cylinders:4, size:2.2}}, 2]
myCar[0].color = red newCar[0].color = red
The new color of my Honda is purple
myCar[0].color = purple
newCar[0].color = purple
```

sort

Sorts the elements of an array.

Method of	Array

Implemented in	JavaScript 1.1, NES 2.0 JavaScript 1.2: modified behavior.
ECMA version	ECMA-262

Syntax
sort(*compareFunction*)

Parameters

compareFunction	Specifies a function that defines the sort order. If omitted, the array is sorted lexicographically (in dictionary order) according to the string conversion of each element.

Description
If compareFunction is not supplied, elements are sorted by converting them to strings and comparing strings in lexicographic ("dictionary" or "telephone book," *not* numerical) order. For example, "80" comes before "9" in lexicographic order, but in a numeric sort 9 comes before 80.

If compareFunction is supplied, the array elements are sorted according to the return value of the compare function. If a and b are two elements being compared, then:

-
- If compareFunction(a, b) is less than 0, sort b to a lower index than a.
- If compareFunction(a, b) returns 0, leave a and b unchanged with respect to each other, but sorted with respect to all different elements.
- If compareFunction(a, b) is greater than 0, sort b to a higher index than a.

So, the compare function has the following form:

```
function compare(a, b) {
  if (a is less than b by some ordering criterion)
    return -1
  if (a is greater than b by the ordering criterion)
    return 1
  // a must be equal to b
  return 0
}
```

To compare numbers instead of strings, the compare function can simply subtract b from a:

```
function compareNumbers(a, b) {
  return a - b
}
```

JavaScript uses a stable sort: the index partial order of a and b does not change if a and b are equal. If a's index was less than b's before sorting, it will be after sorting, no matter how a and b move due to sorting.

The behavior of the sort method changed between JavaScript 1.1 and JavaScript 1.2.

In JavaScript 1.1, on some platforms, the sort method does not work. This method works on all platforms for JavaScript 1.2.

In JavaScript 1.2, this method no longer converts undefined elements to null; instead it sorts them to the high end of the array. For example, assume you have this script:

```
<SCRIPT>
a = new Array();
a[0] = "Ant";
a[5] = "Zebra";

function writeArray(x) {
  for (i = 0; i < x.length; i++) {
    document.write(x[i]);
    if (i < x.length-1) document.write(", ");
  }
}

writeArray(a);
```

```
a.sort();
document.write("<BR><BR>");
writeArray(a);
</SCRIPT>
```

In JavaScript 1.1, JavaScript prints:

```
ant, null, null, null, null, zebra
ant, null, null, null, null, zebra
```

In JavaScript 1.2, JavaScript prints:

```
ant, undefined, undefined, undefined, undefined, zebra
ant, zebra, undefined, undefined, undefined, undefined
```

Examples

The following example creates four arrays and displays the original array, then the sorted arrays. The numeric arrays are sorted without, then with, a compare function.

```
<SCRIPT>
stringArray = new Array("Blue","Humpback","Beluga")
numericStringArray = new Array("80","9","700")
numberArray = new Array(40,1,5,200)
mixedNumericArray = new Array("80","9","700",40,1,5,200)

function compareNumbers(a, b) {
  return a - b
}

document.write("<B>stringArray:</B> " + stringArray.join() +"<BR>")
document.write("<B>Sorted:</B> " + stringArray.sort() +"<P>")

document.write("<B>numberArray:</B> " + numberArray.join() +"<BR>")
document.write("<B>Sorted without a compare function:</B> " + numberArray.sort()
+"<BR>")
document.write("<B>Sorted with compareNumbers:</B> " +
numberArray.sort(compareNumbers) +"<P>")

document.write("<B>numericStringArray:</B> " + numericStringArray.join()
+"<BR>")
document.write("<B>Sorted without a compare function:</B> " +
numericStringArray.sort() +"<BR>")
document.write("<B>Sorted with compareNumbers:</B> " +
```

```
numericStringArray.sort(compareNumbers) +"<P>")

document.write("<B>mixedNumericArray:</B> " + mixedNumericArray.join()
+"<BR>")
document.write("<B>Sorted without a compare function:</B> " +
mixedNumericArray.sort() +"<BR>")
document.write("<B>Sorted with compareNumbers:</B> " +
mixedNumericArray.sort(compareNumbers) +"<BR>")
</SCRIPT>
```

This example produces the following output. As the output shows, when a compare function is used, numbers sort correctly whether they are numbers or numeric strings.

stringArray: Blue,Humpback,Beluga
Sorted: Beluga,Blue,Humpback

numberArray: 40,1,5,200
Sorted without a compare function: 1,200,40,5
Sorted with compareNumbers: 1,5,40,200

numericStringArray: 80,9,700
Sorted without a compare function: 700,80,9
Sorted with compareNumbers: 9,80,700

mixedNumericArray: 80,9,700,40,1,5,200
Sorted without a compare function: 1,200,40,5,700,80,9
Sorted with compareNumbers: 1,5,9,40,80,200,700

See also
Array.join, Array.reverse

splice

Changes the content of an array, adding new elements while removing old elements.

Method of	Array
Implemented in	JavaScript 1.2, NES 3.0 JavaScript 1.3: returns an array containing the removed elements.
ECMA version	ECMA-262 Edition 3

Syntax

splice(*index*, *howMany*, [element1][, ..., *elementN*])

Parameters

index	Index at which to start changing the array.
howMany	An integer indicating the number of old array elements to remove. If howMany is 0, no elements are removed. In this case, you should specify at least one new element.
element1, ..., element*N*	The elements to add to the array. If you don't specify any elements, splice simply removes elements from the array.

Description

If you specify a different number of elements to insert than the number you're removing, the array will have a different length at the end of the call.

The splice method returns an array containing the removed elements. If only one element is removed, an array of one element is returned.

Backward Compatibility

JavaScript 1.2. The splice method returns the element removed, if only one element is removed (howMany parameter is 1); otherwise, the method returns an array containing the removed elements.

Examples
The following script illustrate the use of splice:

```
<SCRIPT LANGUAGE="JavaScript1.2">

myFish = ["angel", "clown", "mandarin", "surgeon"];
document.writeln("myFish: " + myFish + "<BR>");

removed = myFish.splice(2, 0, "drum");
document.writeln("After adding 1: " + myFish);
document.writeln("removed is: " + removed + "<BR>");

removed = myFish.splice(3, 1)
document.writeln("After removing 1: " + myFish);
document.writeln("removed is: " + removed + "<BR>");

removed = myFish.splice(2, 1, "trumpet")
document.writeln("After replacing 1: " + myFish);
document.writeln("removed is: " + removed + "<BR>");

removed = myFish.splice(0, 2, "parrot", "anemone", "blue")
document.writeln("After replacing 2: " + myFish);
document.writeln("removed is: " + removed);

</SCRIPT>
```

This script displays:

```
myFish: ["angel", "clown", "mandarin", "surgeon"]

After adding 1: ["angel", "clown", "drum", "mandarin", "surgeon"]
removed is: undefined
```

After removing 1: ["angel", "clown", "drum", "surgeon"]
removed is: mandarin

After replacing 1: ["angel", "clown", "trumpet", "surgeon"]
removed is: drum

After replacing 2: ["parrot", "anemone", "blue", "trumpet", "surgeon"]
removed is: ["angel", "clown"]

toSource

Returns a string representing the source code of the array.

Method of	Array
Implemented in	JavaScript 1.3

Syntax
toSource()

Parameters
None

Description
The toSource method returns the following values:

-
- For the built-in Array object, toSource returns the following string indicating that the source code is not available:

```
function Array() {
  [native code]
```

```
}
```

- For instances of Array, toSource returns a string representing the source code.

This method is usually called internally by JavaScript and not explicitly in code. You can call toSource while debugging to examine the contents of an array.

Examples
To examine the source code of an array:

```
alpha = new Array("a", "b", "c")
alpha.toSource() //returns ["a", "b", "c"]
```

See also
Array.toString

toString

Returns a string representing the specified array and its elements.

Method of	Array
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Syntax
toString()

Parameters
None.

Description
The Array object overrides the toString method of Object. For Array objects, the toString method joins the array and returns one string containing each array element separated by commas. For example, the following code creates an array and uses toString to convert the array to a string.

```
var monthNames = new Array("Jan","Feb","Mar","Apr")
myVar=monthNames.toString() // assigns "Jan,Feb,Mar,Apr" to myVar
```

JavaScript calls the toString method automatically when an array is to be represented as a text value or when an array is referred to in a string concatenation.

Backward Compatibility

JavaScript 1.2. When you specify LANGUAGE="JavaScript1.2" in the <SCRIPT> tag, toString returns a string representing the source code of the array. This value is the same as the value returned by the toSource method in JavaScript 1.3 and later versions.

```
<SCRIPT LANGUAGE="JavaScript1.2">
var monthNames = new Array("Jan","Feb","Mar","Apr")
myVar=monthNames.toString() // assigns '["Jan", "Feb", "Mar", "Apr"]'
                            // to myVar
</SCRIPT>
```

See also
Array.toSource

unshift

Adds one or more elements to the beginning of an array and returns the new length of the array.

Method of	Array

Implemented in	JavaScript 1.2, NES 3.0
ECMA version	ECMA-262 Edition 3

Syntax
arrayName.unshift(*element1*,..., *elementN*)

Parameters

element1,..., element*N*	The elements to add to the front of the array.

Example
The following code displays the myFish array before and after adding elements to it.

```
myFish = ["angel", "clown"];
document.writeln("myFish before: " + myFish);
unshifted = myFish.unshift("drum", "lion");
document.writeln("myFish after: " + myFish);
document.writeln("New length: " + unshifted);
```

This example displays the following:

```
myFish before: ["angel", "clown"]
myFish after: ["drum", "lion", "angel", "clown"]
New length: 4
```

See also
pop, push, shift

valueOf

Returns the primitive value of an array.

Method of	Array
Implemented in	JavaScript 1.1
ECMA version	ECMA-262

Syntax
valueOf()

Parameters
None

Description
The Array object inherits the valueOf method of Object. The valueOf method of Array returns the primitive value of an array or the primitive value of its elements as follows:

Object type of element	Data type of returned value
Boolean	Boolean

Number or Date	number
All others	string

This method is usually called internally by JavaScript and not explicitly in code.

See also
Object.valueOf

Previous **Contents** **Index** **Next**

Last Updated **September 28, 2000**

Boolean

The Boolean object is an object wrapper for a boolean value.

Core object	
Implemented in	JavaScript 1.1, NES 2.0 JavaScript 1.3: added toSource method
ECMA version	ECMA-262

Created by

The Boolean constructor:

new Boolean(*value*)

Parameters

value	The initial value of the Boolean object. The value is converted to a boolean value, if necessary. If value is omitted or is 0, -0, null, false, NaN, undefined, or the empty string (""), the object has an initial value of false. All other values, including any object or the string "false", create an object with an initial value of true.

Description
Do not confuse the primitive Boolean values true and false with the true and false values of the Boolean object.

Any object whose value is not undefined or null, including a Boolean object whose value is false, evaluates to true when passed to a conditional statement. For example, the condition in the following if statement evaluates to true:

```
x = new Boolean(false);
if(x) //the condition is true
```

This behavior does not apply to Boolean primitives. For example, the condition in the following if statement evaluates to false:

```
x = false;
if(x) //the condition is false
```

Do not use a Boolean object to convert a non-boolean value to a boolean value. Instead, use Boolean as a function to perform this task:

```
x = Boolean(expression) //preferred
x = new Boolean(expression) //don't use
```

If you specify any object, including a Boolean object whose value is false, as the initial value of a Boolean object, the new Boolean object has a value of true.

```
myFalse=new Boolean(false)   // initial value of false
g=new Boolean(myFalse)       //initial value of true
myString=new String("Hello") // string object
s=new Boolean(myString)      //initial value of true
```

Do not use a Boolean object in place of a Boolean primitive.

Backward Compatibility

JavaScript 1.2 and earlier versions. The Boolean object behaves as follows:

-
- When a Boolean object is used as the condition in a conditional test, JavaScript returns the value of the Boolean object. For example, a Boolean object whose value is false is treated as the primitive value false, and a Boolean object whose

value is true is treated as the primitive value true in conditional tests. If the Boolean object is a false object, the conditional statement evaluates to false.

- You can use a Boolean object in place of a Boolean primitive.

Property Summary

Property	Description
constructor	Specifies the function that creates an object's prototype.
prototype	Defines a property that is shared by all Boolean objects.

Method Summary

Method	Description
toSource	Returns an object literal representing the specified Boolean object; you can use this value to create a new object. Overrides the Object.toSource method.
toString	Returns a string representing the specified object. Overrides the Object.toString method.

valueOf	Returns the primitive value of a Boolean object. Overrides the Object.valueOf method.

In addition, this object inherits the watch and unwatch methods from Object.

Examples
The following examples create Boolean objects with an initial value of false:

```
bNoParam = new Boolean()
bZero = new Boolean(0)
bNull = new Boolean(null)
bEmptyString = new Boolean("")
bfalse = new Boolean(false)
```

The following examples create Boolean objects with an initial value of true:

```
btrue = new Boolean(true)
btrueString = new Boolean("true")
bfalseString = new Boolean("false")
bSuLin = new Boolean("Su Lin")
```

constructor

Specifies the function that creates an object's prototype. Note that the value of this property is a reference to the function itself, not a string containing the function's name.

Property of	Boolean
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description
See Object.constructor.

prototype

Represents the prototype for this class. You can use the prototype to add properties or methods to all instances of a class. For information on prototypes, see Function.prototype.

Property of	Boolean
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

toSource

Returns a string representing the source code of the object.

Method of	Boolean
Implemented in	JavaScript 1.3

Syntax
toSource()

Parameters
None

Description
The toSource method returns the following values:

-
- For the built-in Boolean object, toSource returns the following string indicating that the source code is not available:

```
function Boolean() {
  [native code]
}
```

- For instances of Boolean, toSource returns a string representing the source code.

This method is usually called internally by JavaScript and not explicitly in code.

See also
Object.toSource

toString

Returns a string representing the specified Boolean object.

Method of	Boolean
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Syntax
toString()

Parameters
None.

Description
The Boolean object overrides the toString method of the Object object; it does not inherit Object.toString. For Boolean objects, the toString method returns a string representation of the object.

JavaScript calls the toString method automatically when a Boolean is to be represented as a text value or when a Boolean is referred to in a string concatenation.

For Boolean objects and values, the built-in toString method returns the string "true" or "false" depending on the value of the boolean object. In the following code, flag.toString returns "true".

```
var flag = new Boolean(true)
var myVar=flag.toString()
```

See also
Object.toString

valueOf

Returns the primitive value of a Boolean object.

Method of	Boolean
Implemented in	JavaScript 1.1

ECMA version	ECMA-262

Syntax
valueOf()

Parameters
None

Description
The valueOf method of Boolean returns the primitive value of a Boolean object or literal Boolean as a Boolean data type.

This method is usually called internally by JavaScript and not explicitly in code.

Examples
```
x = new Boolean();
myVar=x.valueOf()      //assigns false to myVar
```

See also
Object.valueOf

Last Updated **May 19, 2003**

Date

Lets you work with dates and times.

Core object	
Implemented in	JavaScript 1.0, NES 2.0 JavaScript 1.1: added prototype property. JavaScript 1.3: removed platform dependencies to provide a uniform behavior across platforms; added ms_num parameter to Date constructor; added getFullYear, setFullYear, getMilliseconds, setMilliseconds, toSource, and UTC methods (such as getUTCDate and setUTCDate).
ECMA version	ECMA-262

Created by
The Date constructor:

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

Versions prior to JavaScript 1.3:

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

Parameters

milliseconds	Integer value representing the number of milliseconds since 1 January 1970 00:00:00.
dateString	String value representing a date. The string should be in a format recognized by the Date.parse method.
yr_num, mo_num, day_num	Integer values representing part of a date. As an integer value, the month is represented by 0 to 11 with 0=January and 11=December.
hr_num, min_num, sec_num, ms_num	Integer values representing part of a date.

Description
If you supply no arguments, the constructor creates a Date object for today's date and time according to local time. If you supply some arguments but not others, the missing arguments are set to 0. If you supply any arguments, you must supply at least the year, month, and day. You can omit the hours, minutes, seconds, and milliseconds.

The date is measured in milliseconds since midnight 01 January, 1970 UTC. A day holds 86,400,000 milliseconds. The Date object range is -100,000,000 days to 100,000,000 days relative to 01 January, 1970 UTC.

The Date object provides uniform behavior across platforms.

The Date object supports a number of UTC (universal) methods, as well as local time methods. UTC, also known as Greenwich Mean Time (GMT), refers to the time as set by the World Time Standard. The local time is the time known to the computer where JavaScript is executed.

For compatibility with millennium calculations (in other words, to take into account the year 2000), you should always specify the year in full; for example, use 1998, not 98. To assist you in specifying the complete year, JavaScript includes the methods getFullYear, setFullYear, getFullUTCYear, and setFullUTCYear.

The following example returns the time elapsed between timeA and timeB in milliseconds.

```
timeA = new Date();
// Statements here to take some action.
timeB = new Date();
timeDifference = timeB - timeA;
```

Backward Compatibility

JavaScript 1.2 and earlier. The Date object behaves as follows:

-
- Dates prior to 1970 are not allowed.
- JavaScript depends on platform-specific date facilities and behavior; the behavior of the Date object varies from platform to platform.

Property Summary

Property	Description
constructor	Specifies the function that creates an object's prototype.

prototype	Allows the addition of properties to a Date object.

Method Summary

Method	Description
getDate	Returns the day of the month for the specified date according to local time.
getDay	Returns the day of the week for the specified date according to local time.
getFullYear	Returns the year of the specified date according to local time.
getHours	Returns the hour in the specified date according to local time.
getMilliseconds	Returns the milliseconds in the specified date according to local time.
getMinutes	Returns the minutes in the specified date according to local time.

getMonth	Returns the month in the specified date according to local time.
getSeconds	Returns the seconds in the specified date according to local time.
getTime	Returns the numeric value corresponding to the time for the specified date according to local time.
getTimezoneOffset	Returns the time-zone offset in minutes for the current locale.
getUTCDate	Returns the day (date) of the month in the specified date according to universal time.
getUTCDay	Returns the day of the week in the specified date according to universal time.
getUTCFullYear	Returns the year in the specified date according to universal time.
getUTCHours	Returns the hours in the specified date according to universal time.
getUTCMilliseconds	Returns the milliseconds in the specified date according to universal time.
getUTCMinutes	Returns the minutes in the specified date according to universal time.

getUTCMonth	Returns the month according in the specified date according to universal time.
getUTCSeconds	Returns the seconds in the specified date according to universal time.
getYear	Returns the year in the specified date according to local time.
parse	Returns the number of milliseconds in a date string since January 1, 1970, 00:00:00, local time.
setDate	Sets the day of the month for a specified date according to local time.
setFullYear	Sets the full year for a specified date according to local time.
setHours	Sets the hours for a specified date according to local time.
setMilliseconds	Sets the milliseconds for a specified date according to local time.
setMinutes	Sets the minutes for a specified date according to local time.

setMonth	Sets the month for a specified date according to local time.
setSeconds	Sets the seconds for a specified date according to local time.
setTime	Sets the value of a Date object according to local time.
setUTCDate	Sets the day of the month for a specified date according to universal time.
setUTCFullYear	Sets the full year for a specified date according to universal time.
setUTCHours	Sets the hour for a specified date according to universal time.
setUTCMilliseconds	Sets the milliseconds for a specified date according to universal time.
setUTCMinutes	Sets the minutes for a specified date according to universal time.
setUTCMonth	Sets the month for a specified date according to universal time.
setUTCSeconds	Sets the seconds for a specified date according to universal time.
setYear	Sets the year for a specified date according to local time.

toGMTString	Converts a date to a string, using the Internet GMT conventions.
toLocaleString	Converts a date to a string, using the current locale's conventions.
toLocaleDateString	Returns the "date" portion of the Date as a string, using the current locale's conventions.
toLocaleTimeString	Returns the "time" portion of the Date as a string, using the current locale's conventions.
toSource	Returns an object literal representing the specified Date object; you can use this value to create a new object. Overrides the Object.toSource method.
toString	Returns a string representing the specified Date object. Overrides the Object.toString method.
toUTCString	Converts a date to a string, using the universal time convention.
UTC	Returns the number of milliseconds in a Date object since January 1, 1970, 00:00:00, universal time.
valueOf	Returns the primitive value of a Date object. Overrides the Object.valueOf method.

In addition, this object inherits the watch and unwatch methods from Object.

Examples
The following examples show several ways to assign dates:

```
today = new Date()
birthday = new Date("December 17, 1995 03:24:00")
birthday = new Date(95,11,17)
birthday = new Date(95,11,17,3,24,0)
```

constructor

Specifies the function that creates an object's prototype. Note that the value of this property is a reference to the function itself, not a string containing the function's name.

Property of	Date
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description
See Object.constructor.

getDate

Returns the day of the month for the specified date according to local time.

Method of	Date
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
getDate()

Parameters
None

Description
The value returned by getDate is an integer between 1 and 31.

Examples
The second statement below assigns the value 25 to the variable day, based on the value of the Date object Xmas95.

```
Xmas95 = new Date("December 25, 1995 23:15:00")
day = Xmas95.getDate()
```

See also
Date.getUTCDate, Date.getUTCDay, Date.setDate

getDay

Returns the day of the week for the specified date according to local time.

Method of	Date
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
getDay()

Parameters
None

Description
The value returned by getDay is an integer corresponding to the day of the week: 0 for Sunday, 1 for Monday, 2 for Tuesday, and so on.

Examples
The second statement below assigns the value 1 to weekday, based on the value of the Date object Xmas95. December 25, 1995, is a Monday.

```
Xmas95 = new Date("December 25, 1995 23:15:00")
weekday = Xmas95.getDay()
```

See also
Date.getUTCDay, Date.setDate

getFullYear

Returns the year of the specified date according to local time.

Method of	Date
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax
getFullYear()

Parameters
None

Description
The value returned by getFullYear is an absolute number. For dates between the years 1000 and 9999, getFullYear returns a four-digit number, for example, 1995. Use this function to make sure a year is compliant with years after 2000.

Use this method instead of the getYear method.

Examples
The following example assigns the four-digit value of the current year to the variable yr.

```
var yr;
Today = new Date();
yr = Today.getFullYear();
```

See also
Date.getYear, Date.getUTCFullYear , Date.setFullYear

getHours

Returns the hour for the specified date according to local time.

Method of	Date
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
getHours()

Parameters
None

Description
The value returned by getHours is an integer between 0 and 23.

Examples
The second statement below assigns the value 23 to the variable hours, based on the value of the Date object Xmas95.

```
Xmas95 = new Date("December 25, 1995 23:15:00")
hours = Xmas95.getHours()
```

See also
Date.getUTCHours, Date.setHours

getMilliseconds

Returns the milliseconds in the specified date according to local time.

Method of	Date
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax
getMilliseconds()

Parameters
None

Description
The value returned by getMilliseconds is a number between 0 and 999.

Examples
The following example assigns the milliseconds portion of the current time to the variable ms.

```
var ms;
Today = new Date();
ms = Today.getMilliseconds();
```

See also
Date.getUTCMilliseconds , Date.setMilliseconds

getMinutes

Returns the minutes in the specified date according to local time.

Method of	Date
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
getMinutes()

Parameters
None

Description
The value returned by getMinutes is an integer between 0 and 59.

Examples
The second statement below assigns the value 15 to the variable minutes, based on the value of the Date object Xmas95.

```
Xmas95 = new Date("December 25, 1995 23:15:00")
minutes = Xmas95.getMinutes()
```

See also
Date.getUTCMinutes, Date.setMinutes

getMonth

Returns the month in the specified date according to local time.

Method of	Date
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
getMonth()

Parameters
None

Description
The value returned by getMonth is an integer between 0 and 11. 0 corresponds to January, 1 to February, and so on.

Examples
The second statement below assigns the value 11 to the variable month, based on the value of the Date object Xmas95.

```
Xmas95 = new Date("December 25, 1995 23:15:00")
month = Xmas95.getMonth()
```

See also
Date.getUTCMonth, Date.setMonth

getSeconds

Returns the seconds in the current time according to local time.

Method of	Date
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
getSeconds()

Parameters
None

Description
The value returned by getSeconds is an integer between 0 and 59.

Examples
The second statement below assigns the value 30 to the variable secs, based on the value of the Date object Xmas95.

```
Xmas95 = new Date("December 25, 1995 23:15:30")
secs = Xmas95.getSeconds()
```

See also
Date.getUTCSeconds, Date.setSeconds

getTime

Returns the numeric value corresponding to the time for the specified date according to local time.

Method of	Date
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
getTime()

Parameters
None

Description
The value returned by the getTime method is the number of milliseconds since 1 January 1970 00:00:00. You can use this method to help assign a date and time to another Date object.

Examples
The following example assigns the date value of theBigDay to sameAsBigDay:

```
theBigDay = new Date("July 1, 1999")
sameAsBigDay = new Date()
sameAsBigDay.setTime(theBigDay.getTime())
```

See also
Date.getUTCHours, Date.setTime

getTimezoneOffset

Returns the time-zone offset in minutes for the current locale.

Method of	Date
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
getTimezoneOffset()

Parameters
None

Description
The time-zone offset is the difference between local time and Greenwich Mean Time (GMT). Daylight savings time prevents this value from being a constant.

Examples
x = new Date()
currentTimeZoneOffsetInHours = x.getTimezoneOffset()/60

getUTCDate

Returns the day (date) of the month in the specified date according to universal time.

Method of	Date
Implemented in	JavaScript 1.3

ECMA version	ECMA-262

Syntax
getUTCDate()

Parameters
None

Description
The value returned by getUTCDate is an integer between 1 and 31.

Examples
The following example assigns the day portion of the current date to the variable d.

```
var d;
Today = new Date();
d = Today.getUTCDate();
```

See also
Date.getDate , Date.getUTCDay , Date.setUTCDate

getUTCDay

Returns the day of the week in the specified date according to universal time.

Method of	Date
Implemented in	JavaScript 1.3

ECMA version	ECMA-262

Syntax
getUTCDay()

Parameters
None

Description
The value returned by getUTCDay is an integer corresponding to the day of the week: 0 for Sunday, 1 for Monday, 2 for Tuesday, and so on.

Examples
The following example assigns the weekday portion of the current date to the variable ms.

```
var weekday;
Today = new Date()
weekday = Today.getUTCDay()
```

See also
Date.getDay , Date.getUTCDate , Date.setUTCDate

getUTCFullYear

Returns the year in the specified date according to universal time.

Method of	Date
Implemented in	JavaScript 1.3

ECMA version	ECMA-262

Syntax
getUTCFullYear()

Parameters
None

Description
The value returned by getUTCFullYear is an absolute number that is compliant with year-2000, for example, 1995.

Examples
The following example assigns the four-digit value of the current year to the variable yr.

```
var yr;
Today = new Date();
yr = Today.getUTCFullYear();
```

See also
Date.getFullYear , Date.setFullYear

getUTCHours

Returns the hours in the specified date according to universal time.

Method of	Date
Implemented in	JavaScript 1.3

ECMA version	ECMA-262

Syntax
getUTCHours()

Parameters
None

Description
The value returned by getUTCHours is an integer between 0 and 23.

Examples
The following example assigns the hours portion of the current time to the variable hrs.

```
var hrs;
Today = new Date();
hrs = Today.getUTCHours();
```

See also
Date.getHours , Date.setUTCHours

getUTCMilliseconds

Returns the milliseconds in the specified date according to universal time.

Method of	Date
Implemented in	JavaScript 1.3

ECMA version	ECMA-262

Syntax
getUTCMilliSeconds()

Parameters
None

Description
The value returned by getUTCMilliseconds is an integer between 0 and 999.

Examples
The following example assigns the milliseconds portion of the current time to the variable ms.

```
var ms;
Today = new Date();
ms = Today.getUTCMilliseconds();
```

See also
Date.getMilliseconds , Date.setUTCMilliseconds

getUTCMinutes

Returns the minutes in the specified date according to universal time.

Method of	Date
Implemented in	JavaScript 1.3

ECMA version	ECMA-262

Syntax
getUTCMinutes()

Parameters
None

Description
The value returned by getUTCMinutes is an integer between 0 and 59.

Examples
The following example assigns the minutes portion of the current time to the variable min.

```
var min;
Today = new Date();
min = Today.getUTCMinutes();
```

See also
Date.getMinutes , Date.setUTCMinutes

getUTCMonth

Returns the month according in the specified date according to universal time.

Method of	Date
Implemented in	JavaScript 1.3

ECMA version	ECMA-262

Syntax
getUTCMonth()

Parameters
None

Description
The value returned by getUTCMonth is an integer between 0 and 11 corresponding to the month. 0 for January, 1 for February, 2 for March, and so on.

Examples
The following example assigns the month portion of the current date to the variable mon.

```
var mon;
Today = new Date();
mon = Today.getUTCMonth();
```

See also
Date.getMonth , Date.setUTCMonth

font face="Arial, Helvetica, sans-serif" size= "4">**getUTCSeconds**

Returns the seconds in the specified date according to universal time.

Method of	Date
Implemented in	JavaScript 1.3

ECMA version	ECMA-262

Syntax
getUTCSeconds()

Parameters
None

Description
The value returned by getUTCSeconds is an integer between 0 and 59.

Examples
The following example assigns the seconds portion of the current time to the variable sec.

```
var sec;
Today = new Date();
sec = Today.getUTCSeconds();
```

See also
Date.getSeconds , Date.setUTCSeconds

getYear

Returns the year in the specified date according to local time.

Method of	Date

Implemented in	JavaScript 1.0, NES 2.0 JavaScript 1.3: deprecated; also, getYear returns the year minus 1900 regardless of the year specified
ECMA version	ECMA-262

Syntax
getYear()

Parameters
None

Description
getYear is no longer used and has been replaced by the getFullYear method.

The getYear method returns the year minus 1900; thus:

-
- For years above 2000, the value returned by getYear is 100 or greater. For example, if the year is 2026, getYear returns 126.
- For years between and including 1900 and 1999, the value returned by getYear is between 0 and 99. For example, if the year is 1976, getYear returns 76.
- For years less than 1900 or greater than 1999, the value returned by getYear is less than 0. For example, if the year is 1800, getYear returns -100.

To take into account years before and after 2000, you should use Date.getFullYear instead of getYear so that the year is specified in full.

Backward Compatibility

JavaScript 1.2 and earlier versions. The getYear method returns either a 2-digit or 4-digit year:

-

- For years between and including 1900 and 1999, the value returned by getYear is the year minus 1900. For example, if the year is 1976, the value returned is 76.

- For years less than 1900 or greater than 1999, the value returned by getYear is the four-digit year. For example, if the year is 1856, the value returned is 1856. If the year is 2026, the value returned is 2026.

Examples
Example 1. The second statement assigns the value 95 to the variable year.

```
Xmas = new Date("December 25, 1995 23:15:00")
year = Xmas.getYear() // returns 95
```

Example 2. The second statement assigns the value 100 to the variable year.

```
Xmas = new Date("December 25, 2000 23:15:00")
year = Xmas.getYear() // returns 100
```

Example 3. The second statement assigns the value -100 to the variable year.

```
Xmas = new Date("December 25, 1800 23:15:00")
year = Xmas.getYear() // returns -100
```

Example 4. The second statement assigns the value 95 to the variable year, representing the year 1995.

```
Xmas.setYear(95)
year = Xmas.getYear() // returns 95
```

See also
Date.getFullYear, Date.getUTCFullYear, Date.setYear

parse

Returns the number of milliseconds in a date string since January 1, 1970, 00:00:00, local time.

Method of	Date
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
Date.parse(*dateString*)

Parameters
:

dateString	A string representing a date.

Description
The parse method takes a date string (such as "Dec 25, 1995") and returns the number of milliseconds since January 1, 1970, 00:00:00 (local time). This function is useful for setting date values based on string values, for example in conjunction with the setTime method and the Date object.

Given a string representing a time, parse returns the time value. It accepts the IETF standard date syntax: "Mon, 25 Dec 1995 13:30:00 GMT". It understands the continental US time-zone abbreviations, but for general use, use a time-zone offset, for example, "Mon, 25 Dec 1995 13:30:00 GMT+0430" (4 hours, 30 minutes west of the Greenwich meridian). If you do not specify a time zone, the local time zone is assumed. GMT and UTC are considered equivalent.

Because parse is a static method of Date, you always use it as Date.parse(), rather than as a method of a Date object you created.

Examples
If IPOdate is an existing Date object, then you can set it to August 9, 1995 as follows:

```
IPOdate.setTime(Date.parse("Aug 9, 1995"))
```

See also
Date.UTC

prototype

Represents the prototype for this class. You can use the prototype to add properties or methods to all instances of a class. For information on prototypes, see Function.prototype.

Property of	Date
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

setDate

Sets the day of the month for a specified date according to local time.

Method of	Date
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
setDate(*dayValue*)

Parameters

dayValue	An integer from 1 to 31, representing the day of the month.

Examples
The second statement below changes the day for theBigDay to July 24 from its original value.

```
theBigDay = new Date("July 27, 1962 23:30:00")
theBigDay.setDate(24)
```

See also
Date.getDate, Date.setUTCDate

setFullYear

Sets the full year for a specified date according to local time.

Method of	Date
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax
setFullYear(*yearValue*[, *monthValue*[, *dayValue*]])

Parameters

yearValue	An integer specifying the numeric value of the year, for example, 1995.
monthValue	An integer between 0 and 11 representing the months January through December.
dayValue	An integer between 1 and 31 representing the day of the month. If you specify the dayValue parameter, you must also specify the monthValue.

Description
If you do not specify the monthValue and dayValue parameters, the values returned from the getMonth and getDate methods are used.

If a parameter you specify is outside of the expected range, setFullYear attempts to update the other parameters and the date information in the Date object accordingly. For example, if you specify 15 for monthValue, the year is incremented by 1 (year + 1), and 3 is used for the month.

Examples

```
theBigDay = new Date();
theBigDay.setFullYear(1997);
```

See also

Date.getUTCFullYear ,Date.setUTCFullYear, Date.setYear

setHours

Sets the hours for a specified date according to local time.

Method of	Date
Implemented in	JavaScript 1.0, NES 2.0 JavaScript 1.3: Added minutesValue, secondsValue, and msValue parameters.
ECMA version	ECMA-262

Syntax

setHours(*hoursValue*[, *minutesValue*[, *secondsValue*[, *msValue*]]])

Versions prior to JavaScript 1.3:

setHours(*hoursValue*)

Parameters

hoursValue	An integer between 0 and 23, representing the hour.
minutesValue	An integer between 0 and 59, representing the minutes.
secondsValue	An integer between 0 and 59, representing the seconds. If you specify the secondsValue parameter, you must also specify the minutesValue.
msValue	A number between 0 and 999, representing the milliseconds. If you specify the msValue parameter, you must also specify the minutesValue and secondsValue.

Description

If you do not specify the minutesValue, secondsValue, and msValue parameters, the values returned from the getUTCMinutes, getUTCSeconds, and getMilliseconds methods are used.

If a parameter you specify is outside of the expected range, setHours attempts to update the date information in the Date object accordingly. For example, if you use 100 for secondsValue, the minutes will be incremented by 1 (min + 1), and 40 will be used for seconds.

Examples

theBigDay.setHours(7)

See also

Date.getHours, Date.setUTCHours

setMilliseconds

Sets the milliseconds for a specified date according to local time.

Method of	Date
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax
setMilliseconds(*millisecondsValue*)

Parameters

millisecondsValue	A number between 0 and 999, representing the milliseconds.

Description
If you specify a number outside the expected range, the date information in the Date object is updated accordingly. For example, if you specify 1005, the number of seconds is incremented by 1, and 5 is used for the milliseconds.

Examples

```
theBigDay = new Date();
theBigDay.setMilliseconds(100);
```

See also

Date.getMilliseconds , Date.setUTCMilliseconds

setMinutes

Sets the minutes for a specified date according to local time.

Method of	Date
Implemented in	JavaScript 1.0, NES 2.0 JavaScript 1.3: Added secondsValue and msValue parameters.
ECMA version	ECMA-262

Syntax

setMinutes(*minutesValue*[, *secondsValue*[, *msValue*]])

Versions prior to JavaScript 1.3:

setMinutes(minutesValue)

Parameters

minutesValue	An integer between 0 and 59, representing the minutes.
secondsValue	An integer between 0 and 59, representing the seconds. If you specify the secondsValue parameter, you must also specify the minutesValue.
msValue	A number between 0 and 999, representing the milliseconds. If you specify the msValue parameter, you must also specify the minutesValue and secondsValue.

Examples
theBigDay.setMinutes(45)

Description
If you do not specify the secondsValue and msValue parameters, the values returned from getSeconds and getMilliseconds methods are used.

If a parameter you specify is outside of the expected range, setMinutes attempts to update the date information in the Date object accordingly. For example, if you use 100 for secondsValue, the minutes (minutesValue) will be incremented by 1 (minutesValue + 1), and 40 will be used for seconds.

See also
Date.getMinutes, Date.setUTCMilliseconds

setMonth

Sets the month for a specified date according to local time.

Method of	Date
Implemented in	JavaScript 1.0, NES 2.0 JavaScript 1.3: Added dayValue parameter.
ECMA version	ECMA-262

Syntax
setMonth(*monthValue*[, *dayValue*])

Versions prior to JavaScript 1.3:

setMonth(*monthValue*)

Parameters

monthValue	An integer between 0 and 11 (representing the months January through December).
dayValue	An integer from 1 to 31, representing the day of the month.

Description
If you do not specify the dayValue parameter, the value returned from the getDate method is used.

If a parameter you specify is outside of the expected range, setMonth attempts to update the date information in the Date object accordingly. For example, if you use 15 for

monthValue, the year will be incremented by 1 (year + 1), and 3 will be used for month.

Examples
theBigDay.setMonth(6)

See also
Date.getMonth, Date.setUTCMonth

setSeconds

Sets the seconds for a specified date according to local time.

Method of	Date
Implemented in	JavaScript 1.0, NES 2.0 JavaScript 1.3: Added msValue parameter.
ECMA version	ECMA-262

Syntax
setSeconds(*secondsValue*[, *msValue*])

Versions prior to JavaScript 1.3:

setSeconds(*secondsValue*)

Parameters

secondsValue	An integer between 0 and 59.
msValue	A number between 0 and 999, representing the milliseconds.

Description
If you do not specify the msValue parameter, the value returned from the getMilliseconds methods is used.

If a parameter you specify is outside of the expected range, setSeconds attempts to update the date information in the Date object accordingly. For example, if you use 100 for secondsValue, the minutes stored in the Date object will be incremented by 1, and 40 will be used for seconds.

Examples
theBigDay.setSeconds(30)

See also
Date.getSeconds, Date.setUTCSeconds

setTime

Sets the value of a Date object according to local time.

Method of	Date

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
setTime(*timevalue*)

Parameters

timevalue	An integer representing the number of milliseconds since 1 January 1970 00:00:00.

Description
Use the setTime method to help assign a date and time to another Date object.

Examples
```
theBigDay = new Date("July 1, 1999")
sameAsBigDay = new Date()
sameAsBigDay.setTime(theBigDay.getTime())
```

See also
Date.getTime, Date.setUTCHours

setUTCDate

Sets the day of the month for a specified date according to universal time.

Method of	Date
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax
setUTCDate(*dayValue*)

Parameters

dayValue	An integer from 1 to 31, representing the day of the month.

Description
If a parameter you specify is outside of the expected range, setUTCDate attempts to update the date information in the Date object accordingly. For example, if you use 40 for dayValue, and the month stored in the Date object is June, the day will be changed to 10 and the month will be incremented to July.

Examples
```
theBigDay = new Date();
theBigDay.setUTCDate(20);
```

See also
Date.getUTCDate , Date.setDate

setUTCFullYear

Sets the full year for a specified date according to universal time.

Method of	Date
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax
setUTCFullYear(*yearValue*[, *monthValue*[, *dayValue*]])

Parameters

yearValue	An integer specifying the numeric value of the year, for example, 1995.
monthValue	An integer between 0 and 11 representing the months January through December.
dayValue	An integer between 1 and 31 representing the day of the month. If you specify the dayValue parameter, you must also specify the monthValue.

Description
If you do not specify the monthValue and dayValue parameters, the values returned from the getMonth and getDate methods are used.

If a parameter you specify is outside of the expected range, setUTCFullYear attempts to update the other parameters and the date information in the Date object accordingly. For example, if you specify 15 for monthValue, the year is incremented by 1 (year + 1), and 3 is used for the month.

Examples
theBigDay = new Date();
theBigDay.setUTCFullYear(1997);

See also
Date.getUTCFullYear , Date.setFullYear

setUTCHours

Sets the hour for a specified date according to universal time.

Method of	Date
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax
setUTCHours(*hoursValue*[, *minutesValue*[, *secondsValue*[, *msValue*]]])

Parameters

hoursValue	An integer between 0 and 23, representing the hour.
minutesValue	An integer between 0 and 59, representing the minutes.
secondsValue	An integer between 0 and 59, representing the seconds. If you specify the secondsValue parameter, you must also specify the minutesValue.
msValue	A number between 0 and 999, representing the milliseconds. If you specify the msValue parameter, you must also specify the minutesValue and secondsValue.

Description

If you do not specify the minutesValue, secondsValue, and msValue parameters, the values returned from the getUTCMinutes, getUTCSeconds, and getUTCMilliseconds methods are used.

If a parameter you specify is outside of the expected range, setUTCHours attempts to update the date information in the Date object accordingly. For example, if you use 100 for secondsValue, the minutes will be incremented by 1 (min + 1), and 40 will be used for seconds.

Examples

```
theBigDay = new Date();
theBigDay.setUTCHours(8);
```

See also

Date.getUTCHours , Date.setHours

setUTCMilliseconds

Sets the milliseconds for a specified date according to universal time.

Method of	Date
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax
setUTCMilliseconds(*millisecondsValue*)

Parameters

millisecondsValue	A number between 0 and 999, representing the milliseconds.

Description
If a parameter you specify is outside of the expected range, setUTCMilliseconds attempts to update the date information in the Date object accordingly. For example, if you use 1100 for millisecondsValue, the seconds stored in the Date object will be incremented by 1, and 100 will be used for milliseconds.

Examples

```
theBigDay = new Date();
theBigDay.setUTCMilliseconds(500);
```

See also

Date.getUTCMilliseconds , Date.setMilliseconds

setUTCMinutes

Sets the minutes for a specified date according to universal time.

Method of	Date
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax

setUTCMinutes(*minutesValue*[, *secondsValue*[, *msValue*]])

Parameters

minutesValue	An integer between 0 and 59, representing the minutes.

secondsValue	An integer between 0 and 59, representing the seconds. If you specify the secondsValue parameter, you must also specify the minutesValue.
msValue	A number between 0 and 999, representing the milliseconds. If you specify the msValue parameter, you must also specify the minutesValue and secondsValue.

Description
If you do not specify the secondsValue and msValue parameters, the values returned from getUTCSeconds and getUTCMilliseconds methods are used.

If a parameter you specify is outside of the expected range, setUTCMinutes attempts to update the date information in the Date object accordingly. For example, if you use 100 for secondsValue, the minutes (minutesValue) will be incremented by 1 (minutesValue + 1), and 40 will be used for seconds.

Examples
```
theBigDay = new Date();
theBigDay.setUTCMinutes(43);
```

See also
Date.getUTCMinutes , Date.setMinutes

setUTCMonth

Sets the month for a specified date according to universal time.

Method of	Date

Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax
setUTCMonth(*monthValue*[, *dayValue*])

Parameters

monthValue	An integer between 0 and 11, representing the months January through December.
dayValue	An integer from 1 to 31, representing the day of the month.

Description
If you do not specify the dayValue parameter, the value returned from the getUTCDate method is used.

If a parameter you specify is outside of the expected range, setUTCMonth attempts to update the date information in the Date object accordingly. For example, if you use 15 for monthValue, the year will be incremented by 1 (year + 1), and 3 will be used for month.

Examples
```
theBigDay = new Date();
theBigDay.setUTCMonth(11);
```

See also
Date.getUTCMonth , Date.setMonth

setUTCSeconds

Sets the seconds for a specified date according to universal time.

Method of	Date
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax
setUTCSeconds(*secondsValue*[, *msValue*])

Parameters

secondsValue	An integer between 0 and 59.
msValue	A number between 0 and 999, representing the milliseconds.

Description
If you do not specify the msValue parameter, the value returned from the getUTCMilliseconds methods is used.

If a parameter you specify is outside of the expected range, setUTCSeconds attempts to update the date information in the Date object accordingly. For example, if you use 100 for secondsValue, the minutes stored in the Date object will be incremented by 1, and 40 will be used for seconds.

Examples
theBigDay = new Date();
theBigDay.setUTCSeconds(20);

See also
Date.getUTCSeconds , Date.setSeconds

setYear

Sets the year for a specified date according to local time.

Method of	Date
Implemented in	JavaScript 1.0, NES 2.0 Deprecated in JavaScript 1.3.
ECMA version	ECMA-262

Syntax
setYear(*yearValue*)

Parameters

yearValue	An integer.

Description
setYear is no longer used and has been replaced by the setFullYear method.

If yearValue is a number between 0 and 99 (inclusive), then the year for dateObjectName is set to 1900 + yearValue. Otherwise, the year for dateObjectName is set to yearValue.

To take into account years before and after 2000, you should use setFullYear instead of setYear so that the year is specified in full.

Examples
Note that there are two ways to set years in the 20th century.

Example 1. The year is set to 1996.

theBigDay.setYear(96)

Example 2. The year is set to 1996.

theBigDay.setYear(1996)

Example 3. The year is set to 2000.

theBigDay.setYear(2000)

See also
Date.getYear, Date.setFullYear, Date.setUTCFullYear

toGMTString

Converts a date to a string, using the Internet GMT conventions.

Method of	Date
Implemented in	JavaScript 1.0, NES 2.0 Deprecated in JavaScript 1.3.
ECMA version	ECMA-262

Syntax
toGMTString()

Parameters
None

Description
toGMTString is no longer used and has been replaced by the toUTCString method.

The exact format of the value returned by toGMTString varies according to the platform.

You should use Date.toUTCString instead of toGMTSTring.

Examples
In the following example, today is a Date object:

today.toGMTString()

In this example, the toGMTString method converts the date to GMT (UTC) using the operating system's time-zone offset and returns a string value that is similar to the

following form. The exact format depends on the platform.

Mon, 18 Dec 1995 17:28:35 GMT

See also
Date.toLocaleString, Date.toUTCString

toLocaleString

Converts a date to a string, using the current locale's conventions.

Method of	Date
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
toLocaleString()

Parameters
None

Description
The toLocaleString method relies on the underlying operating system in formatting dates. It converts the date to a string using the formatting convention of the operating system where the script is running. For example, in the United States, the month appears before the date (04/15/98), whereas in Germany the date appears before the month (15.04.98). If the operating system is not year-2000 compliant and does not use the full year for years before 1900 or over 2000, toLocaleString returns a string that is not year-2000 compliant. toLocaleString behaves similarly to toString when converting a year that the operating system does not properly format.

Methods such as getHours, getMinutes, and getSeconds give more portable results than toLocaleString.

Examples
In the following example, today is a Date object:

today = new Date(95,11,18,17,28,35) //months are represented by 0 to 11
today.toLocaleString()

In this example, toLocaleString returns a string value that is similar to the following form. The exact format depends on the platform.

12/18/95 17:28:35

See also
Date.toGMTString, Date.toUTCString

toLocaleDateString

Converts a date to a string, returning the "date" portion using the current locale's conventions.

Method of	Date
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
toLocaleDateString()

Parameters
None

Description
The toLocaleDateString method relies on the underlying operating system in formatting dates. It converts the date to a string using the formatting convention of the operating system where the script is running. For example, in the United States, the month appears before the date (04/15/98), whereas in Germany the date appears before the month (15.04.98). If the operating system is not year-2000 compliant and does not use the full year for years before 1900 or over 2000, toLocaleDateString returns a string that is not year-2000 compliant. toLocaleDateString behaves similarly to toString when converting a year that the operating system does not properly format.

Methods such as getHours, getMinutes, and getSeconds give more portable results than toLocaleDateString.

Examples
In the following example, today is a Date object:

```
today = new Date(95,11,18,17,28,35) //months are represented by 0 to 11
today.toLocaleDateString()
```

In this example, toLocaleDateString returns a string value that is similar to the following form. The exact format depends on the platform.

12/18/95

See also
Date.toGMTString, Date.toUTCString

toLocaleTimeString

Converts a date to a string, returning the "date" portion using the current locale's conventions.

Method of	Date
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
toLocaleTimeString()

Parameters
None

Description
The toLocaleTimeString method relies on the underlying operating system in formatting dates. It converts the date to a string using the formatting convention of the operating system where the script is running. For example, in the United States, the month appears before the date (04/15/98), whereas in Germany the date appears before the month (15.04.98). If the operating system is not year-2000 compliant and does not use the full year for years before 1900 or over 2000, toLocaleTimeString returns a string that is not year-2000 compliant. toLocaleTimeString behaves similarly to toString when converting a year that the operating system does not properly format.

Methods such as getHours, getMinutes, and getSeconds give more portable results than toLocaleTimeString.

Examples
In the following example, today is a Date object:

```
today = new Date(95,11,18,17,28,35) //months are represented by 0 to 11
today.toLocaleTimeString()
```

In this example, toLocaleTimeString returns a string value that is similar to the following form. The exact format depends on the platform.

17:28:35

See also
Date.toGMTString, Date.toUTCString

toSource

Returns a string representing the source code of the object.

Method of	Date
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax
toSource()

Parameters
None

Description
The toSource method returns the following values:

-
- For the built-in Date object, toSource returns the following string indicating that the source code is not available:

```
function Date() {
  [native code]
}
```

- For instances of Date, toSource returns a string representing the source code.

This method is usually called internally by JavaScript and not explicitly in code.

See also
Object.toSource

toString

Returns a string representing the specified Date object.

Method of	Date
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Syntax
toString()

Parameters
None.

Description
The Date object overrides the toString method of the Object object; it does not inherit Object.toString. For Date objects, the toString method returns a string representation of the object.

JavaScript calls the toString method automatically when a date is to be represented as a text value or when a date is referred to in a string concatenation.

Examples

The following example assigns the toString value of a Date object to myVar:

```
x = new Date();
myVar=x.toString();   //assigns a value to myVar similar to:
    //Mon Sep 28 14:36:22 GMT-0700 (Pacific Daylight Time) 1998
```

See also
Object.toString

toUTCString

Converts a date to a string, using the universal time convention.

Method of	Date
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax
toUTCString()

Parameters
None

Description
The value returned by toUTCString is a readable string formatted according to UTC convention. The format of the return value may vary according to the platform.

Examples
var UTCstring;

```
Today = new Date();
UTCstring = Today.toUTCString();
```

See also
Date.toLocaleString, Date.toUTCString

UTC

Returns the number of milliseconds in a Date object since January 1, 1970, 00:00:00, universal time.

Method of	Date
Static	
Implemented in	JavaScript 1.0, NES 2.0 JavaScript 1.3: added ms parameter.
ECMA version	ECMA-262

Syntax
Date.UTC(*year*, *month*[, *day*[, *hrs*[, *min*[, *sec*[, *ms*]]]]])

Parameters

year	A year after 1900.
month	An integer between 0 and 11 representing the month.
date	An integer between 1 and 31 representing the day of the month.
hrs	An integer between 0 and 23 representing the hours.
min	An integer between 0 and 59 representing the minutes.
sec	An integer between 0 and 59 representing the seconds.
ms	An integer between 0 and 999 representing the milliseconds.

Description

UTC takes comma-delimited date parameters and returns the number of milliseconds between January 1, 1970, 00:00:00, universal time and the time you specified.

You should specify a full year for the year; for example, 1998. If a year between 0 and 99 is specified, the method converts the year to a year in the 20th century (1900 + year); for example, if you specify 95, the year 1995 is used.

The UTC method differs from the Date constructor in two ways.

-
- Date.UTC uses universal time instead of the local time.
- Date.UTC returns a time value as a number instead of creating a Date object.

If a parameter you specify is outside of the expected range, the UTC method updates the other parameters to allow for your number. For example, if you use 15 for month, the year will be incremented by 1 (year + 1), and 3 will be used for the month.

Because UTC is a static method of Date, you always use it as Date.UTC(), rather than as a method of a Date object you created.

Examples
The following statement creates a Date object using GMT instead of local time:

```
gmtDate = new Date(Date.UTC(96, 11, 1, 0, 0, 0))
```

See also
Date.parse

valueOf

Returns the primitive value of a Date object.

Method of	Date
Implemented in	JavaScript 1.1
ECMA version	ECMA-262

Syntax
valueOf()

Parameters
None

Description
The valueOf method of Date returns the primitive value of a Date object as a number data type, the number of milliseconds since midnight 01 January, 1970 UTC.

This method is usually called internally by JavaScript and not explicitly in code.

Examples
```
x = new Date(56,6,17);
myVar=x.valueOf()      //assigns -424713600000 to myVar
```

See also
Object.valueOf

Previous **Contents** **Index** **Next**

Last Updated **May 19, 2003**

Function

xxx I believe that I have removed all client-specific examples from this file.

Specifies a string of JavaScript code to be compiled as a function.

Core object	
Implemented in	JavaScript 1.1, NES 2.0 JavaScript 1.2: added arity, arguments.callee properties; added ability to nest functions. JavaScript 1.3: added apply, call, and toSource methods; deprecated arguments.caller property. JavaScript 1.4: deprecated arguments, arguments.callee, arguments.length, and arity properties (arguments remains a variable local to a function rather than a property of Function).
ECMA version	ECMA-262

Created by
The Function constructor:

new Function ([*arg1*[, *arg2*[, ... *argN*]],] *functionBody*)

The function statement (see function for details):

function *name*([*param*[, *param*[, ... *param*]]]) {
  statements
}

Parameters

arg1, arg2, ... arg*N*	Names to be used by the function as formal argument names. Each must be a string that corresponds to a valid JavaScript identifier; for example "x" or "theValue".
functionBody	A string containing the JavaScript statements comprising the function definition.
name	The function name.
param	The name of an argument to be passed to the function. A function can have up to 255 arguments.
statements	The statements comprising the body of the function.

Description

Function objects created with the Function constructor are evaluated each time they are used. This is less efficient than declaring a function and calling it within your code, because declared functions are compiled.

To return a value, the function must have a return statement that specifies the value to

return.

All parameters are passed to functions *by value*; the value is passed to the function, but if the function changes the value of the parameter, this change is not reflected globally or in the calling function. However, if you pass an object as a parameter to a function and the function changes the object's properties, that change is visible outside the function, as shown in the following example:

```
function myFunc(theObject) {
  theObject.make="Toyota"
}

mycar = {make:"Honda", model:"Accord", year:1998}
x=mycar.make     // returns Honda
myFunc(mycar)    // pass object mycar to the function
y=mycar.make     // returns Toyota (prop was changed by the function)
```

The this keyword does not refer to the currently executing function, so you must refer to Function objects by name, even within the function body.

Accessing a function's arguments with the arguments array. You can refer to a function's arguments within the function by using the arguments array. See arguments.

Specifying arguments with the Function constructor. The following code creates a Function object that takes two arguments.

```
var multiply = new Function("x", "y", "return x * y")
```

The arguments "x" and "y" are formal argument names that are used in the function body, "return x * y".

The preceding code assigns a function to the variable multiply. To call the Function object, you can specify the variable name as if it were a function, as shown in the following examples.

```
var theAnswer = multiply(7,6)

var myAge = 50
if (myAge >=39) {myAge=multiply (myAge,.5)}
```

Assigning a function to a variable with the Function constructor. Suppose you create the variable multiply using the Function constructor, as shown in the preceding section:

```
var multiply = new Function("x", "y", "return x * y")
```

This is similar to declaring the following function:

```
function multiply(x,y) {
  return x*y
}
```

Assigning a function to a variable using the Function constructor is similar to declaring a function with the function statement, but they have differences:

-
- When you assign a function to a variable using var multiply = new Function("..."), multiply is a variable for which the current value is a reference to the function created with new Function().

- When you create a function using function multiply() {...}, multiply is not a variable, it is the name of a function.

Nesting functions. You can nest a function within a function. The nested (inner) function is private to its containing (outer) function:

-
- The inner function can be accessed only from statements in the outer function.

- The inner function can use the arguments and variables of the outer function. The outer function cannot use the arguments and variables of the inner function.

The following example shows nested functions:

```
function addSquares (a,b) {
  function square(x) {
    return x*x
  }
  return square(a) + square(b)
}
a=addSquares(2,3) // returns 13
b=addSquares(3,4) // returns 25
c=addSquares(4,5) // returns 41
```

When a function contains a nested function, you can call the outer function and specify arguments for both the outer and inner function:

```
function outside(x) {
  function inside(y) {
    return x+y
  }
  return inside
}
result=outside(3)(5) // returns 8
```

Specifying an event handler with a Function object. The following code assigns a function to a window's onFocus event handler (the event handler must be spelled in all lowercase):

```
window.onfocus = new Function("document.bgColor='antiquewhite'")
```

If a function is assigned to a variable, you can assign the variable to an event handler. The following code assigns a function to the variable setBGColor.

```
var setBGColor = new Function("document.bgColor='antiquewhite'")
```

You can use this variable to assign a function to an event handler in either of the following ways:

```
document.form1.colorButton.onclick=setBGColor
```

```
<INPUT NAME="colorButton" TYPE="button"
  VALUE="Change background color"
  onClick="setBGColor()">
```

Once you have a reference to a Function object, you can use it like a function and it will convert from an object to a function:

```
window.onfocus()
```

Event handlers do not take arguments, so you cannot declare any arguments in a Function constructor for an event handler. For example, you cannot call the function multiply by setting a button's onclick property as follows:

```
document.form1.button1.onclick=multFun(5,10)
```

Backward Compatibility

JavaScript 1.1 and earlier versions. You cannot nest a function statement in another

statement or in itself.

Property Summary

Property	Description
arguments	An array corresponding to the arguments passed to a function.
arguments.callee	Specifies the function body of the currently executing function.
arguments.caller	Specifies the name of the function that invoked the currently executing function.
arguments.length	Specifies the number of arguments passed to the function.
arity	Specifies the number of arguments expected by the function.
constructor	Specifies the function that creates an object's prototype.

length	Specifies the number of arguments expected by the function.
prototype	Allows the addition of properties to a Function object.

Method Summary

Method	Description
apply	Allows you to apply a method of another object in the context of a different object (the calling object).
call	Allows you to call (execute) a method of another object in the context of a different object (the calling object).
toSource	Returns a string representing the source code of the function. Overrides the Object.toSource method.
toString	Returns a string representing the source code of the function. Overrides the Object.toString method.
valueOf	Returns a string representing the source code of the function. Overrides the Object.valueOf method.

Examples

Example 1. The following function returns a string containing the formatted representation of a number padded with leading zeros.

```
// This function returns a string padded with leading zeros
function padZeros(num, totalLen) {
  var numStr = num.toString()           // Initialize return value
                                        // as string
  var numZeros = totalLen - numStr.length // Calculate no. of zeros
  if (numZeros > 0) {
    for (var i = 1; i <= numZeros; i++) {
      numStr = "0" + numStr
    }
  }
  return numStr
}
```

The following statements call the padZeros function.

```
result=padZeros(42,4) // returns "0042"
result=padZeros(42,2) // returns "42"
result=padZeros(5,4)  // returns "0005"
```

Example 2. You can determine whether a function exists by comparing the function name to null. In the following example, func1 is called if the function noFunc does not exist; otherwise func2 is called. Notice that the window name is needed when referring to the function name noFunc.

```
if (window.noFunc == null)
  func1()
else func2()
```

Example 3. The following example creates onFocus and onBlur event handlers for a frame. This code exists in the same file that contains the FRAMESET tag. Note that this is the only way to create onFocus and onBlur event handlers for a frame, because you cannot specify the event handlers in the FRAME tag.

```
frames[0].onfocus = new Function("document.bgColor='antiquewhite'")
frames[0].onblur = new Function("document.bgColor='lightgrey'")
```

apply

This feature is not in the ECMA specification that corresponds to JavaScript 1.3, but is expected in the next version.

Allows you to apply a method of another object in the context of a different object (the calling object).

Method of	Function
Implemented in	JavaScript 1.3

Syntax

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

Parameters

thisArg	Parameter for the calling object
argArray	An argument array for the object

Description

You can assign a different this object when calling an existing function. this refers to the current object, the calling object. With apply, you can write a method once and then inherit it in another object, without having to rewrite the method for the new object.

apply is very similar to call, except for the type of arguments it supports. You can use an arguments array instead of a named set of parameters. With apply, you can use an array literal, for example, apply(this, [name, value]), or an Array object, for example, apply(this, new Array(name, value)).

You can also use arguments for the argArray parameter. arguments is a local variable of a function. It can be used for all unspecified arguments of the called object. Thus, you do not have to know the arguments of the called object when you use the apply method. You can use arguments to pass all the arguments to the called object. The called object is then responsible for handling the arguments.

Examples

You can use apply to chain constructors for an object, similar to Java. In the following example, the constructor for the product object is defined with two parameters, name and value. Another object, prod_dept, initializes its unique variable (dept) and calls the constructor for product in its constructor to initialize the other variables. In this example, the parameter arguments is used for all arguments of the product object's constructor.

```
function product(name, value){
  this.name = name;
  if(value > 1000)
    this.value = 999;
  else
    this.value = value;
}

function prod_dept(name, value, dept){
  this.dept = dept;
  product.apply(product, arguments);
}

prod_dept.prototype = new product();

// since 5 is less than 100 value is set
cheese = new prod_dept("feta", 5, "food");

// since 5000 is above 1000, value will be 999
car = new prod_dept("honda", 5000, "auto");
```

See also

Function.call

arguments

An array corresponding to the arguments passed to a function.

Local variable of	All function objects
Property of	Function (deprecated)
Implemented in	JavaScript 1.1, NES 2.0 JavaScript 1.2: added arguments.callee property. JavaScript 1.3: deprecated arguments.caller property; removed support for argument names and local variable names as properties of the arguments array. JavaScript 1.4: deprecated arguments, arguments.callee, and arguments.length as properties of Function; retained arguments as a local variable of a function and arguments.callee and arguments.length as properties of this variable.
ECMA version	ECMA-262

Description

The arguments array is a local variable available within all function objects; arguments as a property of Function is no longer used.

You can refer to a function's arguments within the function by using the arguments array. This array contains an entry for each argument passed to the function. For example, if a function is passed three arguments, you can refer to the arguments as

follows:

arguments[0]
arguments[1]
arguments[2]

The arguments array is available only within a function body. Attempting to access the arguments array outside a function declaration results in an error.

You can use the arguments array if you call a function with more arguments than it is formally declared to accept. This technique is useful for functions that can be passed a variable number of arguments. You can use arguments.length to determine the number of arguments passed to the function, and then process each argument by using the arguments array. (To determine the number of arguments declared when a function was defined, use the Function.length property.)

The arguments array has the following properties:

Property	Description
arguments.callee	Specifies the function body of the currently executing function.
arguments.caller	Specifies the name of the function that invoked the currently executing function. (Deprecated)
arguments.length	Specifies the number of arguments passed to the function.

Backward Compatibility

JavaScript 1.3 and earlier versions. In addition to being available as a local variable, the arguments array is also a property of the Function object and can be preceded by the function name. For example, if a function myFunc is passed three arguments named

arg1, arg2, and arg3, you can refer to the arguments as follows:

myFunc.arguments[0]
myFunc.arguments[1]
myFunc.arguments[2]

JavaScript 1.1 and 1.2. The following features, which were available in JavaScript 1.1 and JavaScript 1.2, have been removed:

-
- Each local variable of a function is a property of the arguments array. For example, if a function myFunc has a local variable named myLocalVar, you can refer to the variable as arguments.myLocalVar.

- Each formal argument of a function is a property of the arguments array. For example, if a function myFunc has two arguments named arg1 and arg2, you can refer to the arguments as arguments.arg1 and arguments.arg2. (You can also refer to them as arguments[0] and arguments[1].)

Examples
Example 1. This example defines a function that concatenates several strings. The only formal argument for the function is a string that specifies the characters that separate the items to concatenate. The function is defined as follows:

```
function myConcat(separator) {
  result="" // initialize list
  // iterate through arguments
  for (var i=1; i<arguments.length; i++) {
    result += arguments[i] + separator
  }
  return result
}
```

You can pass any number of arguments to this function, and it creates a list using each argument as an item in the list.

```
// returns "red, orange, blue, "
myConcat(", ","red","orange","blue")

// returns "elephant; giraffe; lion; cheetah;"
myConcat("; ","elephant","giraffe","lion", "cheetah")
```

```
// returns "sage. basil. oregano. pepper. parsley. "
myConcat(". ","sage","basil","oregano", "pepper", "parsley")
```

Example 2. This example defines a function that creates HTML lists. The only formal argument for the function is a string that is "U" if the list is to be unordered (bulleted), or "O" if the list is to be ordered (numbered). The function is defined as follows:

```
function list(type) {
  document.write("<" + type + "L>") // begin list
  // iterate through arguments
  for (var i=1; i<arguments.length; i++) {
    document.write("<LI>" + arguments[i])
  }
  document.write("</" + type + "L>") // end list
}
```

You can pass any number of arguments to this function, and it displays each argument as an item in the type of list indicated. For example, the following call to the function

```
list("U", "One", "Two", "Three")
```

results in this output:

```
<UL>
<LI>One
<LI>Two
<LI>Three
</UL>
```

In server-side JavaScript, you can display the same output by calling the write function instead of using document.write.

arguments.callee

Specifies the function body of the currently executing function.

Property of	arguments local variable; Function (deprecated)
Implemented in	JavaScript 1.2 JavaScript 1.4: Deprecated callee as a property of Function.arguments, retained it as a property of a function's local arguments variable.
ECMA version	ECMA-262

Description

arguments.callee is a property of the arguments local variable available within all function objects; arguments.callee as a property of Function is no longer used.

The callee property is available only within the body of a function.

The this keyword does not refer to the currently executing function. Use the callee property to refer to a function within the function body.

Examples

The following function returns the value of the function's callee property.

```
function myFunc() {
  return arguments.callee
}
```

The following value is returned:

```
function myFunc() { return arguments.callee; }
```

See also

Function.arguments

arguments.caller

Specifies the name of the function that invoked the currently executing function.

Property of	Function
Implemented in	JavaScript 1.1, NES 2.0 Deprecated in JavaScript 1.3

Description
caller is no longer used.

The caller property is available only within the body of a function.

If the currently executing function was invoked by the top level of a JavaScript program, the value of caller is null.

The this keyword does not refer to the currently executing function, so you must refer to functions and Function objects by name, even within the function body.

The caller property is a reference to the calling function, so

-
- If you use it in a string context, you get the result of calling functionName.toString. That is, the decompiled canonical source form of the function.

- You can also call the calling function, if you know what arguments it might want. Thus, a called function can call its caller without knowing the name of the particular caller, provided it knows that all of its callers have the same form and fit, and that they will not call the called function again unconditionally (which would result in infinite recursion).

Examples
The following code checks the value of a function's caller property.

```
function myFunc() {
```

```
  if (arguments.caller == null) {
    return ("The function was called from the top!")
  } else return ("This function's caller was " + arguments.caller)
}
```

See also
Function.arguments

arguments.length

Specifies the number of arguments passed to the function.

Property of	arguments local variable; Function (deprecated)
Implemented in	JavaScript 1.1 JavaScript 1.4: Deprecated length as a property of Function.arguments, retained it as a property of a function's local arguments variable.
ECMA version	ECMA-262

Description
arguments.length is a property of the arguments local variable available within all function objects; arguments.length as a property of Function is no longer used.

arguments.length provides the number of arguments actually passed to a function. By contrast, the Function.length property indicates how many arguments a function expects.

Example
The following example demonstrates the use of Function.length and arguments.length.

```
function addNumbers(x,y){
  if (arguments.length == addNumbers.length) {
    return (x+y)
  }
  else return 0
}
```

If you pass more than two arguments to this function, the function returns 0:

```
result=addNumbers(3,4,5)   // returns 0
result=addNumbers(3,4)     // returns 7
result=addNumbers(103,104) // returns 207
```

See also
Function.arguments

arity

Specifies the number of arguments expected by the function.

Property of	Function
Implemented in	JavaScript 1.2, NES 3.0 Deprecated in JavaScript 1.4.

Description
arity is no longer used and has been replaced by the length property.

arity is external to the function, and indicates how many arguments a function expects. By contrast, arguments.length provides the number of arguments actually passed to a function.

Example
The following example demonstrates the use of arity and arguments.length.

```
function addNumbers(x,y){
  if (arguments.length == addNumbers.length) {
    return (x+y)
  }
  else return 0
}
```

If you pass more than two arguments to this function, the function returns 0:

```
result=addNumbers(3,4,5)   // returns 0
result=addNumbers(3,4)     // returns 7
result=addNumbers(103,104) // returns 207
```

See also
arguments.length, Function.length

call

This feature is not in the ECMA specification that corresponds to JavaScript 1.3, but is expected in the next version.

Allows you to call (execute) a method of another object in the context of a different object (the calling object).

Method of	Function
Implemented in	JavaScript 1.3

Syntax
call(*thisArg*[, *arg1*[, *arg2*[, ...]]])

Parameters

thisArg	Parameter for the calling object
arg1, arg2, ...	Arguments for the object

Description
You can assign a different this object when calling an existing function. this refers to the current object, the calling object.

With call, you can write a method once and then inherit it in another object, without having to rewrite the method for the new object.

Examples
You can use call to chain constructors for an object, similar to Java. In the following example, the constructor for the product object is defined with two parameters, name and value. Another object, prod_dept, initializes its unique variable (dept) and calls the constructor for product in its constructor to initialize the other variables.

```
function product(name, value){
  this.name = name;
  if(value > 1000)
    this.value = 999;
  else
    this.value = value;
}

function prod_dept(name, value, dept){
  this.dept = dept;
  product.call(this, name, value);
```

```
}

prod_dept.prototype = new product();

// since 5 is less than 100 value is set
cheese = new prod_dept("feta", 5, "food");

// since 5000 is above 1000, value will be 999
car = new prod_dept("honda", 5000, "auto");
```

See also
Function.apply

constructor

Specifies the function that creates an object's prototype. Note that the value of this property is a reference to the function itself, not a string containing the function's name.

Property of	Function
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description
See Object.constructor.

length

Specifies the number of arguments expected by the function.

Property of	Function
Implemented in	JavaScript 1.1
ECMA version	ECMA-262

Description
length is external to a function, and indicates how many arguments the function expects. By contrast, arguments.length is local to a function and provides the number of arguments actually passed to the function.

Example
See the example for arguments.length.

See also
arguments.length

prototype

A value from which instances of a particular class are created. Every object that can be created by calling a constructor function has an associated prototype property.

Property of	Function
Implemented in	JavaScript 1.1, NES 2.0

ECMA version	ECMA-262

Description

You can add new properties or methods to an existing class by adding them to the prototype associated with the constructor function for that class. The syntax for adding a new property or method is:

fun.prototype.*name* = *value*

where

fun	The name of the constructor function object you want to change.
name	The name of the property or method to be created.
value	The value initially assigned to the new property or method.

If you add a property to the prototype for an object, then all objects created with that object's constructor function will have that new property, even if the objects existed before you created the new property. For example, assume you have the following statements:

```
var array1 = new Array();
var array2 = new Array(3);
Array.prototype.description=null;
array1.description="Contains some stuff"
array2.description="Contains other stuff"
```

After you set a property for the prototype, all subsequent objects created with Array will have the property:

```
anotherArray=new Array()
anotherArray.description="Currently empty"
```

Example

The following example creates a method, str_rep, and uses the statement String.prototype.rep = str_rep to add the method to all String objects. All objects created with new String() then have that method, even objects already created. The example then creates an alternate method and adds that to one of the String objects using the statement s1.rep = fake_rep. The str_rep method of the remaining String objects is not altered.

```
var s1 = new String("a")
var s2 = new String("b")
var s3 = new String("c")

// Create a repeat-string-N-times method for all String objects
function str_rep(n) {
  var s = "", t = this.toString()
  while (--n >= 0) s += t
  return s
}

String.prototype.rep = str_rep

s1a=s1.rep(3) // returns "aaa"
s2a=s2.rep(5) // returns "bbbbb"
s3a=s3.rep(2) // returns "cc"

// Create an alternate method and assign it to only one String variable
function fake_rep(n) {
  return "repeat " + this + " " + n + " times."
}

s1.rep = fake_rep
s1b=s1.rep(1) // returns "repeat a 1 times."
s2b=s2.rep(4) // returns "bbbb"
s3b=s3.rep(6) // returns "cccccc"
```

The function in this example also works on String objects not created with the String

constructor. The following code returns "zzz".

```
"z".rep(3)
```

toSource

This feature is not in the ECMA specification that corresponds to JavaScript 1.3, but is expected in the next version.

Returns a string representing the source code of the function.

Method of	Function
Implemented in	JavaScript 1.3

Syntax
toSource()

Parameters
None

Description
The toSource method returns the following values:

-
- For the built-in Function object, toSource returns the following string indicating that the source code is not available:

```
function Function() {
  [native code]
}
```

- For custom functions, toSource returns the JavaScript source that defines the

object as a string.

This method is usually called internally by JavaScript and not explicitly in code. You can call toSource while debugging to examine the contents of an object.

See also
Function.toString, Object.valueOf

toString

Returns a string representing the source code of the function.

Method of	Function
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Syntax
toString()

Parameters
None.

Description
The Function object overrides the toString method of the Object object; it does not inherit Object.toString. For Function objects, the toString method returns a string representation of the object.

JavaScript calls the toString method automatically when a Function is to be represented as a text value or when a Function is referred to in a string concatenation.

For Function objects, the built-in toString method decompiles the function back into the JavaScript source that defines the function. This string includes the function keyword, the argument list, curly braces, and function body.

For example, assume you have the following code that defines the Dog object type and creates theDog, an object of type Dog:

```
function Dog(name,breed,color,sex) {
  this.name=name
  this.breed=breed
  this.color=color
  this.sex=sex
}

theDog = new Dog("Gabby","Lab","chocolate","girl")
```

Any time Dog is used in a string context, JavaScript automatically calls the toString function, which returns the following string:

```
function Dog(name, breed, color, sex) { this.name = name; this.breed = breed; this.color = color; this.sex = sex; }
```

See also
Object.toString

valueOf

Returns a string representing the source code of the function.

Method of	Function
Implemented in	JavaScript 1.1

ECMA version	ECMA-262

Syntax
valueOf()

Parameters
None

Description
The valueOf method returns the following values:

-
- For the built-in Function object, valueOf returns the following string indicating that the source code is not available:

```
function Function() {
  [native code]
}
```

- For custom functions, toSource returns the JavaScript source that defines the object as a string. The method is equivalent to the toString method of the function.

This method is usually called internally by JavaScript and not explicitly in code.

See also
Function.toString, Object.valueOf

Last Updated **September 28, 2000**

java

A top-level object used to access any Java class in the package java.*.

Core object	
Implemented in	JavaScript 1.1, NES 2.0

Created by

The java object is a top-level, predefined JavaScript object. You can automatically access it without using a constructor or calling a method.

Description

The java object is a convenience synonym for the property Packages.java.

See also

Packages, Packages.java

Last Updated **September 28, 2000**

JavaArray

A wrapped Java array accessed from within JavaScript code is a member of the type JavaArray.

Core object	
Implemented in	JavaScript 1.1, NES 2.0

Created by

Any Java method which returns an array. In addition, you can create a JavaArray with an arbitrary data type using the newInstance method of the Array class:

```
public static Object newInstance(Class componentType,
  int length)
  throws NegativeArraySizeException
```

Description

The JavaArray object is an instance of a Java array that is created in or passed to JavaScript. JavaArray is a wrapper for the instance; all references to the array instance are made through the JavaArray.

In JavaScript 1.4 and later, the componentType parameter is either a JavaClass object representing the type of the array or class object, such as one returned by java.lang.Class.forName. In JavaScript 1.3 and earlier, componentType must be a class object.

Use zero-based indexes to access the elements in a JavaArray object, just as you do to access elements in an array in Java. For example:

```
var javaString = new java.lang.String("Hello world!");
var byteArray = javaString.getBytes();
byteArray[0] // returns 72
byteArray[1] // returns 101
```

Any Java data brought into JavaScript is converted to JavaScript data types. When the JavaArray is passed back to Java, the array is unwrapped and can be used by Java code. See the *Core JavaScript Guide* for more information about data type conversions.

In JavaScript 1.4 and later, the methods of java.lang.Object are inherited by JavaArray.

Backward compatibility

JavaScript 1.3 and earlier. The methods of java.lang.Object are not inherited by JavaArray. In addition, the toString method is inherited from the Object object and returns the following value:

[object JavaArray]

You must specify a class object, such as one returned by java.lang.Object.forName, for the componentType parameter of newInstance when you use this method to create an array. You cannot use a JavaClass object for the componentType parameter.

Property Summary

Property	Description
length	The number of elements in the Java array represented by JavaArray.

Method Summary

Method	Description
toString	In JavaScript 1.4, this method is overridden by the inherited method java.lang.Object.toString. In JavaScript 1.3 and earlier, this method returns a string identifying the object as a JavaArray.

In JavaScript 1.4 and later, JavaArray also inherits methods from the Java array superclass, java.lang.Object.

Examples
Example 1. Instantiating a JavaArray in JavaScript.

In this example, the JavaArray byteArray is created by the java.lang.String.getBytes method, which returns an array.

```
var javaString = new java.lang.String("Hello world!");
var byteArray = javaString.getBytes();
```

Example 2. Instantiating a JavaArray in JavaScript with the newInstance method.

In JavaScript 1.4, you can use a JavaClass object as the argument for the newInstance method which creates the array, as shown in the following code:

```
var dogs = java.lang.reflect.Array.newInstance(java.lang.String, 5)
```

In JavaScript 1.1, use a class object returned by java.lang.Class.forName as the argument for the newInstance method, as shown in the following code:

```
var dataType = java.lang.Class.forName("java.lang.String")
var dogs = java.lang.reflect.Array.newInstance(dataType, 5)
```

length

The number of elements in the Java array represented by the JavaArray object.

Property of	JavaArray
Implemented in	JavaScript 1.1, NES 2.0

Description
Unlike Array.length, JavaArray.length is a read-only property. You cannot change the value of the JavaArray.length property because Java arrays have a fixed number of elements.

See also
Array.length

toString

Returns a string representation of the JavaArray.

Method of	JavaArray
Implemented in	JavaScript 1.1, NES 2.0

Parameters
None

Description
Calls the method java.lang.Object.toString, which returns the value of the following expression:

```
JavaArray.getClass().getName() + '@' +
    java.lang.Integer.toHexString(JavaArray.hashCode())
```

Backward compatibility

JavaScript 1.3 and earlier. The toString method is inherited from the Object object and returns the following value:

[object JavaArray]

Previous **Contents** **Index** **Next**

Last Updated **September 28, 2000**

JavaClass

A JavaScript reference to a Java class.

Core object	
Implemented in	JavaScript 1.1, NES 2.0

Created by

A reference to the class name used with the Packages object:

Packages.*JavaClass*

where *JavaClass* is the fully-specified name of the object's Java class. The LiveConnect java, sun, and netscape objects provide shortcuts for commonly used Java packages and also create JavaClass objects.

Description

A JavaClass object is a reference to one of the classes in a Java package, such as netscape.javascript.JSObject. A JavaPackage object is a reference to a Java package, such as netscape.javascript. In JavaScript, the JavaPackage and JavaClass hierarchy reflect the Java package and class hierarchy.

You can pass a JavaClass object to a Java method which requires an argument of type java.lang.Class.

Backward compatibility

JavaScript 1.3 and earlier. You must create a wrapper around an instance of java.lang.Class before you pass it as a parameter to a Java method-JavaClass objects are not automatically converted to instances of java.lang.Class.

Property Summary
The properties of a JavaClass object are the static fields of the Java class.

Method Summary
The methods of a JavaClass object are the static methods of the Java class.

Examples
Example 1. In the following example, x is a JavaClass object referring to java.awt.Font. Because BOLD is a static field in the Font class, it is also a property of the JavaClass object.

```
x = java.awt.Font
myFont = x("helv",x.BOLD,10) // creates a Font object
```

The previous example omits the Packages keyword and uses the java synonym because the Font class is in the java package.

Example 2. In the following example, the JavaClass object java.lang.String is passed as an argument to the newInstance method which creates an array:

```
var cars = java.lang.reflect.Array.newInstance(java.lang.String, 15)
```

See also
JavaArray, JavaObject, JavaPackage, Packages

Previous **Contents** **Index** **Next**

Last Updated **September 28, 2000**

JavaObject

The type of a wrapped Java object accessed from within JavaScript code.

Core object	
Implemented in	JavaScript 1.1, NES 2.0

Created by

Any Java method which returns an object type. In addition, you can explicitly construct a JavaObject using the object's Java constructor with the Packages keyword:

new Packages.*JavaClass*(*parameterList*)

where *JavaClass* is the fully-specified name of the object's Java class.

Parameters

parameterList	An optional list of parameters, specified by the constructor in the Java class.

Description

The JavaObject object is an instance of a Java class that is created in or passed to

JavaScript. JavaObject is a wrapper for the instance; all references to the class instance are made through the JavaObject.

Any Java data brought into JavaScript is converted to JavaScript data types. When the JavaObject is passed back to Java, it is unwrapped and can be used by Java code. See the *Core JavaScript Guide* for more information about data type conversions.

Property Summary
Inherits public data members from the Java class of which it is an instance as properties. It also inherits public data members from any superclass as properties.

Method Summary
Inherits public methods from the Java class of which it is an instance. The JavaObject also inherits methods from java.lang.Object and any other superclass.

Examples
Example 1. Instantiating a Java object in JavaScript.

The following code creates the JavaObject theString, which is an instance of the class java.lang.String:

```
var theString = new Packages.java.lang.String("Hello, world")
```

Because the String class is in the java package, you can also use the java synonym and omit the Packages keyword when you instantiate the class:

```
var theString = new java.lang.String("Hello, world")
```

Example 2. Accessing methods of a Java object.

Because the JavaObject theString is an instance of java.lang.String, it inherits all the public methods of java.lang.String. The following example uses the startsWith method to check whether theString begins with "Hello".

```
var theString = new java.lang.String("Hello, world")
theString.startsWith("Hello") // returns true
```

Example 3. Accessing inherited methods.

Because getClass is a method of Object, and java.lang.String extends Object, the String

class inherits the getClass method. Consequently, getClass is also a method of the JavaObject which instantiates String in JavaScript.

```
var theString = new java.lang.String("Hello, world")
theString.getClass() // returns java.lang.String
```

See also
JavaArray, JavaClass, JavaPackage, Packages

Previous **Contents** **Index** **Next**

Last Updated **September 28, 2000**

JavaPackage

A JavaScript reference to a Java package.

Core object	
Implemented in	JavaScript 1.1, NES 2.0

Created by

A reference to the package name used with the Packages keyword:

Packages.*JavaPackage*

where *JavaPackage* is the name of the object's Java package. If the package is in the java, netscape, or sun packages, the Packages keyword is optional.

Description

In Java, a package is a collection of Java classes or other Java packages. For example, the netscape package contains the package netscape.javascript; the netscape.javascript package contains the classes JSObject and JSException.

In JavaScript, a JavaPackage is a reference to a Java package. For example, a reference to netscape is a JavaPackage. netscape.javascript is both a JavaPackage and a property of the netscape JavaPackage.

A JavaClass object is a reference to one of the classes in a package, such as netscape.javascript.JSObject. The JavaPackage and JavaClass hierarchy reflect the Java package and class hierarchy.

Although the packages and classes contained in a JavaPackage are its properties, you cannot use a for...in statement to enumerate them as you can enumerate the properties of other objects.

Property Summary

The properties of a JavaPackage are the JavaClass objects and any other JavaPackage objects it contains.

Examples

Suppose the Redwood corporation uses the Java redwood package to contain various Java classes that it implements. The following code creates the JavaPackage red:

```
var red = Packages.redwood
```

See also

JavaArray, JavaClass, JavaObject, Packages

Previous **Contents** **Index** **Next**

Last Updated **September 28, 2000**

Math

A built-in object that has properties and methods for mathematical constants and functions. For example, the Math object's PI property has the value of pi.

Core object	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Created by

The Math object is a top-level, predefined JavaScript object. You can automatically access it without using a constructor or calling a method.

Description

All properties and methods of Math are static. You refer to the constant PI as Math.PI and you call the sine function as Math.sin(x), where x is the method's argument. Constants are defined with the full precision of real numbers in JavaScript.

It is often convenient to use the with statement when a section of code uses several Math constants and methods, so you don't have to type "Math" repeatedly. For example,

```
with (Math) {
  a = PI * r*r
  y = r*sin(theta)
  x = r*cos(theta)
}
```

Property Summary

Property	Description
E	Euler's constant and the base of natural logarithms, approximately 2.718.
LN2	Natural logarithm of 2, approximately 0.693.
LN10	Natural logarithm of 10, approximately 2.302.
LOG2E	Base 2 logarithm of E (approximately 1.442).
LOG10E	Base 10 logarithm of E (approximately 0.434).
PI	Ratio of the circumference of a circle to its diameter, approximately 3.14159.
SQRT1_2	Square root of 1/2; equivalently, 1 over the square root of 2, approximately 0.707.

SQRT2	Square root of 2, approximately 1.414.

Method Summary

Method	Description
abs	Returns the absolute value of a number.
acos	Returns the arccosine (in radians) of a number.
asin	Returns the arcsine (in radians) of a number.
atan	Returns the arctangent (in radians) of a number.
atan2	Returns the arctangent of the quotient of its arguments.
ceil	Returns the smallest integer greater than or equal to a number.

cos	Returns the cosine of a number.
exp	Returns E^{number}, where number is the argument, and E is Euler's constant, the base of the natural logarithms.
floor	Returns the largest integer less than or equal to a number.
log	Returns the natural logarithm (base E) of a number.
max	Returns the greater of two numbers.
min	Returns the lesser of two numbers.
pow	Returns base to the exponent power, that is, base ^{exponent}.
random	Returns a pseudo-random number between 0 and 1.

round	Returns the value of a number rounded to the nearest integer.
sin	Returns the sine of a number.
sqrt	Returns the square root of a number.
tan	Returns the tangent of a number.

In addition, this object inherits the watch and unwatch methods from Object.

abs

Returns the absolute value of a number.

Method of	Math
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
abs(*x*)

Parameters

x	A number

Examples
The following function returns the absolute value of the variable x:

```
function getAbs(x) {
  return Math.abs(x)
}
```

Description
Because abs is a static method of Math, you always use it as Math.abs(), rather than as a method of a Math object you created.

acos

Returns the arccosine (in radians) of a number.

Method of	Math
Static	

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
acos(*x*)

Parameters

x	A number

Description
The acos method returns a numeric value between 0 and pi radians. If the value of number is outside this range, it returns NaN.

Because acos is a static method of Math, you always use it as Math.acos(), rather than as a method of a Math object you created.

Examples
The following function returns the arccosine of the variable x:

```
function getAcos(x) {
  return Math.acos(x)
}
```

If you pass -1 to getAcos, it returns 3.141592653589793; if you pass 2, it returns NaN because 2 is out of range.

See also

Math.asin, Math.atan, Math.atan2, Math.cos, Math.sin, Math.tan

asin

Returns the arcsine (in radians) of a number.

Method of	Math
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
asin(*x*)

Parameters

x	A number

Description
The asin method returns a numeric value between -pi/2 and pi/2 radians. If the value of number is outside this range, it returns NaN.

Because asin is a static method of Math, you always use it as Math.asin(), rather than as a method of a Math object you created.

Examples
The following function returns the arcsine of the variable x:

```
function getAsin(x) {
  return Math.asin(x)
}
```

If you pass getAsin the value 1, it returns 1.570796326794897 (pi/2); if you pass it the value 2, it returns NaN because 2 is out of range.

See also
Math.acos, Math.atan, Math.atan2, Math.cos, Math.sin, Math.tan

atan

Returns the arctangent (in radians) of a number.

Method of	Math
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
atan(*x*)

Parameters

x	A number

Description

The atan method returns a numeric value between -pi/2 and pi/2 radians.

Because atan is a static method of Math, you always use it as Math.atan(), rather than as a method of a Math object you created.

Examples

The following function returns the arctangent of the variable x:

```
function getAtan(x) {
  return Math.atan(x)
}
```

If you pass getAtan the value 1, it returns 0.7853981633974483; if you pass it the value .5, it returns 0.4636476090008061.

See also

Math.acos, Math.asin, Math.atan2, Math.cos, Math.sin, Math.tan

atan2

Returns the arctangent of the quotient of its arguments.

Method of	Math
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
atan2(*y*, *x*)

Parameters

y, x	Number

Description
The atan2 method returns a numeric value between -pi and pi representing the angle theta of an (x,y) point. This is the counterclockwise angle, measured in radians, between the positive X axis, and the point (x,y). Note that the arguments to this function pass the y-coordinate first and the x-coordinate second.

atan2 is passed separate x and y arguments, and atan is passed the ratio of those two arguments.

Because atan2 is a static method of Math, you always use it as Math.atan2(), rather than as a method of a Math object you created.

Examples

The following function returns the angle of the polar coordinate:

```
function getAtan2(x,y) {
  return Math.atan2(x,y)
}
```

If you pass getAtan2 the values (90,15), it returns 1.4056476493802699; if you pass it the values (15,90), it returns 0.16514867741462683.

See also
Math.acos, Math.asin, Math.atan, Math.cos, Math.sin, Math.tan

ceil

Returns the smallest integer greater than or equal to a number.

Method of	Math
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
ceil(*x*)

Parameters

x	A number

Description
Because ceil is a static method of Math, you always use it as Math.ceil(), rather than as a method of a Math object you created.

Examples
The following function returns the ceil value of the variable x:

```
function getCeil(x) {
  return Math.ceil(x)
}
```

If you pass 45.95 to getCeil, it returns 46; if you pass -45.95, it returns -45.

See also
Math.floor

cos

Returns the cosine of a number.

Method of	Math
Static	
Implemented in	JavaScript 1.0, NES 2.0

ECMA version	ECMA-262

Syntax
cos(*x*)

Parameters

x	A number

Description
The cos method returns a numeric value between -1 and 1, which represents the cosine of the angle.

Because cos is a static method of Math, you always use it as Math.cos(), rather than as a method of a Math object you created.

Examples
The following function returns the cosine of the variable x:

```
function getCos(x) {
  return Math.cos(x)
}
```

If x equals 2*Math.PI, getCos returns 1; if x equals Math.PI, the getCos method returns -1.

See also
Math.acos, Math.asin, Math.atan, Math.atan2, Math.sin, Math.tan

E

Euler's constant and the base of natural logarithms, approximately 2.718.

Property of	Math
Static, Read-only	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Description
Because E is a static property of Math, you always use it as Math.E, rather than as a property of a Math object you created.

Examples
The following function returns Euler's constant:

```
function getEuler() {
  return Math.E
}
```

exp

Returns E^x, where x is the argument, and E is Euler's constant, the base of the natural logarithms.

Method of	Math
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
exp(x)

Parameters

x	A number

Description
Because exp is a static method of Math, you always use it as Math.exp(), rather than as a method of a Math object you created.

Examples
The following function returns the exponential value of the variable x:

```
function getExp(x) {
  return Math.exp(x)
}
```

If you pass getExp the value 1, it returns 2.718281828459045.

See also
Math.E, Math.log, Math.pow

floor

Returns the largest integer less than or equal to a number.

Method of	Math
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
floor(*x*)

Parameters

x	A number

Description

Because floor is a static method of Math, you always use it as Math.floor(), rather than as a method of a Math object you created.

Examples
The following function returns the floor value of the variable x:

```
function getFloor(x) {
  return Math.floor(x)
}
```

If you pass 45.95 to getFloor, it returns 45; if you pass -45.95, it returns -46.

See also
Math.ceil

LN2

The natural logarithm of 2, approximately 0.693.

Property of	Math
Static, Read-only	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Examples
The following function returns the natural log of 2:

```
function getNatLog2() {
```

```
  return Math.LN2
}
```

Description
Because LN2 is a static property of Math, you always use it as Math.LN2, rather than as a property of a Math object you created.

LN10

The natural logarithm of 10, approximately 2.302.

Property of	Math
Static, Read-only	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Examples
The following function returns the natural log of 10:

```
function getNatLog10() {
  return Math.LN10
}
```

Description
Because LN10 is a static property of Math, you always use it as Math.LN10, rather than as a property of a Math object you created.

log

Returns the natural logarithm (base E) of a number.

Method of	Math
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
log(*x*)

Parameters

x	A number

Description
If the value of number is negative, the return value is always NaN.

Because log is a static method of Math, you always use it as Math.log(), rather than as a method of a Math object you created.

Examples
The following function returns the natural log of the variable x:

```
function getLog(x) {
  return Math.log(x)
}
```

If you pass getLog the value 10, it returns 2.302585092994046; if you pass it the value 0, it returns -Infinity; if you pass it the value -1, it returns NaN because -1 is out of range.

See also
Math.exp, Math.pow

LOG2E

The base 2 logarithm of E (approximately 1.442).

Property of	Math
Static, Read-only	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Examples
The following function returns the base 2 logarithm of E:

```
function getLog2e() {
  return Math.LOG2E
}
```

Description
Because LOG2E is a static property of Math, you always use it as Math.LOG2E, rather than as a property of a Math object you created.

LOG10E

The base 10 logarithm of E (approximately 0.434).

Property of	Math
Static, Read-only	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Examples
The following function returns the base 10 logarithm of E:

```
function getLog10e() {
  return Math.LOG10E
}
```

Description
Because LOG10E is a static property of Math, you always use it as Math.LOG10E, rather than as a property of a Math object you created.

max

Returns the larger of two numbers.

Method of	Math
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax

max(*x*,*y*)

Parameters

x, y	Numbers.

Description

Because max is a static method of Math, you always use it as Math.max(), rather than as a method of a Math object you created.

Examples

The following function evaluates the variables x and y:

```
function getMax(x,y) {
```

```
  return Math.max(x,y)
}
```

If you pass getMax the values 10 and 20, it returns 20; if you pass it the values -10 and -20, it returns -10.

See also
Math.min

min

Returns the smaller of two numbers.

Method of	Math
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
min(*x*,*y*)

Parameters

x, y	Numbers

Description
Because min is a static method of Math, you always use it as Math.min(), rather than as a method of a Math object you created.

Examples
The following function evaluates the variables x and y:

```
function getMin(x,y) {
  return Math.min(x,y)
}
```

If you pass getMin the values 10 and 20, it returns 10; if you pass it the values -10 and -20, it returns -20.

See also
Math.max

PI

The ratio of the circumference of a circle to its diameter, approximately 3.14159.

Property of	Math
Static, Read-only	
Implemented in	JavaScript 1.0, NES 2.0

ECMA version	ECMA-262

Examples
The following function returns the value of pi:

```
function getPi() {
  return Math.PI
}
```

Description
Because PI is a static property of Math, you always use it as Math.PI, rather than as a property of a Math object you created.

pow

Returns base to the exponent power, that is, $base^{exponent}$.

Method of	Math
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
pow(*x*,*y*)

Parameters

base	The base number
exponent	The exponent to which to raise base

Description
Because pow is a static method of Math, you always use it as Math.pow(), rather than as a method of a Math object you created.

Examples

```
function raisePower(x,y) {
  return Math.pow(x,y)
}
```

If x is 7 and y is 2, raisePower returns 49 (7 to the power of 2).

See also
Math.exp, Math.log

random

Returns a pseudo-random number between 0 and 1. The random number generator is seeded from the current time, as in Java.

Method of	Math
Static	
Implemented in	JavaScript 1.0, NES 2.0: Unix only JavaScript 1.1, NES 2.0: all platforms
ECMA version	ECMA-262

Syntax
random()

Parameters
None.

Description
Because random is a static method of Math, you always use it as Math.random(), rather than as a method of a Math object you created.

Examples
```
//Returns a random number between 0 and 1
function getRandom() {
  return Math.random()
}
```

round

Returns the value of a number rounded to the nearest integer.

Method of	Math
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
round(*x*)

Parameters

x	A number

Description
If the fractional portion of number is .5 or greater, the argument is rounded to the next higher integer. If the fractional portion of number is less than .5, the argument is rounded to the next lower integer.

Because round is a static method of Math, you always use it as Math.round(), rather than as a method of a Math object you created.

Examples
```
//Returns the value 20
x=Math.round(20.49)

//Returns the value 21
```

```
x=Math.round(20.5)

//Returns the value -20
x=Math.round(-20.5)

//Returns the value -21
x=Math.round(-20.51)
```

sin

Returns the sine of a number.

Method of	Math
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
sin(*x*)

Parameters

x	A number

Description
The sin method returns a numeric value between -1 and 1, which represents the sine of the argument.

Because sin is a static method of Math, you always use it as Math.sin(), rather than as a method of a Math object you created.

Examples
The following function returns the sine of the variable x:

```
function getSine(x) {
  return Math.sin(x)
}
```

If you pass getSine the value Math.PI/2, it returns 1.

See also
Math.acos, Math.asin, Math.atan, Math.atan2, Math.cos, Math.tan

sqrt

Returns the square root of a number.

Method of	Math
Static	

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
sqrt(*x*)

Parameters

x	A number

Description
If the value of number is negative, sqrt returns NaN.

Because sqrt is a static method of Math, you always use it as Math.sqrt(), rather than as a method of a Math object you created.

Examples
The following function returns the square root of the variable x:

```
function getRoot(x) {
  return Math.sqrt(x)
}
```

If you pass getRoot the value 9, it returns 3; if you pass it the value 2, it returns 1.414213562373095.

SQRT1_2

The square root of 1/2; equivalently, 1 over the square root of 2, approximately 0.707.

Property of	Math
Static, Read-only	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Examples
The following function returns 1 over the square root of 2:

```
function getRoot1_2() {
  return Math.SQRT1_2
}
```

Description
Because SQRT1_2 is a static property of Math, you always use it as Math.SQRT1_2, rather than as a property of a Math object you created.

SQRT2

The square root of 2, approximately 1.414.

Property of	Math

Static, Read-only	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Examples
The following function returns the square root of 2:

```
function getRoot2() {
  return Math.SQRT2
}
```

Description
Because SQRT2 is a static property of Math, you always use it as Math.SQRT2, rather than as a property of a Math object you created.

tan

Returns the tangent of a number.

Method of	Math
Static	
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
tan(*x*)

Parameters

x	A number

Description
The tan method returns a numeric value that represents the tangent of the angle.

Because tan is a static method of Math, you always use it as Math.tan(), rather than as a method of a Math object you created.

Examples
The following function returns the tangent of the variable x:

```
function getTan(x) {
  return Math.tan(x)
}
```

See also
Math.acos, Math.asin, Math.atan, Math.atan2, Math.cos, Math.sin

Previous **Contents** **Index** **Next**

Last Updated **September 28, 2000**

netscape

A top-level object used to access any Java class in the package netscape.*.

Core object	
Implemented in	JavaScript 1.1, NES 2.0

Created by
The netscape object is a top-level, predefined JavaScript object. You can automatically access it without using a constructor or calling a method.

Description
The netscape object is a convenience synonym for the property Packages.netscape.

See also
Packages, Packages.netscape

Last Updated **September 28, 2000**

Number

Lets you work with numeric values. The Number object is an object wrapper for primitive numeric values.

Core object	
Implemented in	JavaScript 1.1, NES 2.0 JavaScript 1.2: modified behavior of Number constructor. JavaScript 1.3: added toSource method. JavaScript 1.5, NES 6.0: added toExponential, toFixed, and toPrecision methods.
ECMA version	ECMA-262

Created by

The Number constructor:

new Number(*value*)

Parameters

value	The numeric value of the object being created.

Description
The primary uses for the Number object are:

-
- To access its constant properties, which represent the largest and smallest representable numbers, positive and negative infinity, and the Not-a-Number value.

- To create numeric objects that you can add properties to. Most likely, you will rarely need to create a Number object.

The properties of Number are properties of the class itself, not of individual Number objects.

JavaScript 1.2: Number(x) now produces NaN rather than an error if x is a string that does not contain a well-formed numeric literal. For example,

```
x=Number("three");

document.write(x + "<BR>");
```

prints NaN

You can convert any object to a number using the top-level Number function.

Property Summary

Property	Description

constructor	Specifies the function that creates an object's prototype.
MAX_VALUE	The largest representable number.
MIN_VALUE	The smallest representable number.
NaN	Special "not a number" value.
NEGATIVE_INFINITY	Special value representing negative infinity; returned on overflow.
POSITIVE_INFINITY	Special value representing infinity; returned on overflow.
prototype	Allows the addition of properties to a Number object.

Method Summary

Method	Description

toExponential	Returns a string representing the number in exponential notation.
toFixed	Returns a string representing the number in fixed-point notation.
toPrecision	Returns a string representing the number to a specified precision in fixed-point notation.
toSource	Returns an object literal representing the specified Number object; you can use this value to create a new object. Overrides the Object.toSource method.
toString	Returns a string representing the specified object. Overrides the Object.toString method.
valueOf	Returns the primitive value of the specified object. Overrides the Object.valueOf method.

In addition, this object inherits the watch and unwatch methods from Object.

Examples

Example 1. The following example uses the Number object's properties to assign values to several numeric variables:

```
biggestNum = Number.MAX_VALUE;
smallestNum = Number.MIN_VALUE;
infiniteNum = Number.POSITIVE_INFINITY;
negInfiniteNum = Number.NEGATIVE_INFINITY;
notANum = Number.NaN;
```

Example 2. The following example creates a Number object, myNum, then adds a description property to all Number objects. Then a value is assigned to the myNum object's description property.

```
myNum = new Number(65);
Number.prototype.description=null;
myNum.description="wind speed";
```

constructor

Specifies the function that creates an object's prototype. Note that the value of this property is a reference to the function itself, not a string containing the function's name.

Property of	Number
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description
See Object.constructor.

MAX_VALUE

The maximum numeric value representable in JavaScript.

Property of	Number

Static, Read-only	
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description
The MAX_VALUE property has a value of approximately 1.79E+308. Values larger than MAX_VALUE are represented as "Infinity".

Because MAX_VALUE is a static property of Number, you always use it as Number.MAX_VALUE, rather than as a property of a Number object you created.

Examples
The following code multiplies two numeric values. If the result is less than or equal to MAX_VALUE, the func1 function is called; otherwise, the func2 function is called.

```
if (num1 * num2 <= Number.MAX_VALUE)
  func1()
else
  func2()
```

MIN_VALUE

The smallest positive numeric value representable in JavaScript.

Property of	Number
Static, Read-only	

Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description
The MIN_VALUE property is the number closest to 0, not the most negative number, that JavaScript can represent.

MIN_VALUE has a value of approximately 5e-324. Values smaller than MIN_VALUE ("underflow values") are converted to 0.

Because MIN_VALUE is a static property of Number, you always use it as Number.MIN_VALUE, rather than as a property of a Number object you created.

Examples
The following code divides two numeric values. If the result is greater than or equal to MIN_VALUE, the func1 function is called; otherwise, the func2 function is called.

```
if (num1 / num2 >= Number.MIN_VALUE)
  func1()
else
  func2()
```

NaN

A special value representing Not-A-Number. This value is represented as the unquoted literal NaN.

Property of	Number
Read-only	

Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description
JavaScript prints the value Number.NaN as NaN.

NaN is always unequal to any other number, including NaN itself; you cannot check for the not-a-number value by comparing to Number.NaN. Use the **isNaN** function instead.

You might use the NaN property to indicate an error condition for a function that should return a valid number.

Examples
In the following example, if month has a value greater than 12, it is assigned NaN, and a message is displayed indicating valid values.

```
var month = 13
if (month < 1 || month > 12) {
   month = Number.NaN
   alert("Month must be between 1 and 12.")
}
```

See also
NaN, isNaN, parseFloat, parseInt

NEGATIVE_INFINITY

A special numeric value representing negative infinity. This value is represented as the unquoted literal "-Infinity".

Property of	Number
Static, Read-only	
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description
This value behaves slightly differently than mathematical infinity:

-
- Any positive value, including POSITIVE_INFINITY, multiplied by NEGATIVE_INFINITY is NEGATIVE_INFINITY.
- Any negative value, 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 value except NEGATIVE_INFINITY, is POSITIVE_INFINITY.
- NEGATIVE_INFINITY, divided by any positive value 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.

Because NEGATIVE_INFINITY is a static property of Number, you always use it as Number.NEGATIVE_INFINITY, rather than as a property of a Number object you created.

Examples
In the following example, the variable smallNumber is assigned a value that is smaller than the minimum value. When the if statement executes, smallNumber has the value "-Infinity", so the func1 function is called.

```
var smallNumber = -Number.MAX_VALUE*10
if (smallNumber == Number.NEGATIVE_INFINITY)
  func1()
else
  func2()
```

See also
Infinity, isFinite

POSITIVE_INFINITY

A special numeric value representing infinity. This value is represented as the unquoted literal "Infinity".

Property of	Number
Static, Read-only	
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description
This value behaves slightly differently than mathematical infinity:

-
- Any positive value, including POSITIVE_INFINITY, multiplied by POSITIVE_INFINITY is POSITIVE_INFINITY.

- Any negative value, 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 value except NEGATIVE_INFINITY, is NEGATIVE_INFINITY.
- POSITIVE_INFINITY, divided by any positive value 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.

Because POSITIVE_INFINITY is a static property of Number, you always use it as Number.POSITIVE_INFINITY, rather than as a property of a Number object you created.

Examples
In the following example, the variable bigNumber is assigned a value that is larger than the maximum value. When the if statement executes, bigNumber has the value "Infinity", so the func1 function is called.

```
var bigNumber = Number.MAX_VALUE * 10
if (bigNumber == Number.POSITIVE_INFINITY)
  func1()
else
  func2()
```

See also
Infinity, isFinite

prototype

Represents the prototype for this class. You can use the prototype to add properties or methods to all instances of a class. For information on prototypes, see Function.prototype.

Property of	Number
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

toExponential

Returns a string representing the Number object in exponential notation.

Method of	Number
Implemented in	JavaScript 1.5
ECMA version	ECMA-262, Edition 3

Syntax
toExponential([*fractionDigits*])

Parameters

fractionDigits	An integer specifying the number of digits after the decimal point. Defaults to as many digits as necessary to specify the number.

Description
The Number.prototype.toExponential method returns a string representing a Number object in exponential notation with one digit before the decimal point, rounded to *fractionDigits* digits after the decimal point. If the *fractionDigits* argument is omitted, the number of digits after the decimal point defaults to the number of digits necessary to represent the value uniquely.

If you use the toExponential method for a numeric literal and the numeric literal has no exponent and no decimal point, leave a space before the dot that precedes the method call to prevent the dot from being interpreted as a decimal point.

If a number has more digits that requested by the *fractionDigits* parameter, the number is rounded to the nearest number represented by *fractionDigits* digits. See the discussion of rounding in the description of the toFixed method on page 129, which also applies to toExponential.

Examples
```
var num=77.1234
alert("num.toExponential() is " + num.toExponential()) //displays 7.71234e+1
alert("num.toExponential(4) is " + num.toExponential(4)) //displays 7.7123e+1
alert("num.toExponential(2) is " + num.toExponential(2)) //displays 7.71e+1
alert("77.1234.toExponential() is " + 77.1234.toExponential())
//displays 7.71234e+1
alert("77 .toExponential() is " + 77 .toExponential()) //displays 7.7e+1
```

See also
toFixed, toPrecision, toString

toFixed

Returns a string representing the Number object in fixed-point notation.

Method of	Number
Implemented in	JavaScript 1.5
ECMA version	ECMA-262, Edition 3

Syntax
toFixed([*fractionDigits*])

Parameters

fractionDigits	An integer specifying the number of digits after the decimal point. Defaults to zero.

Description
The Number.prototype.toFixed method returns a string representing a Number object in fixed-point notation, rounded to the number of digits after the decimal point specified by *fractionDigits*.

The output of toFixed may be more precise than toString for some values, because toString outputs only enough significant digits to distinguish the number from adjacent number values.

If a number has more digits that requested by the *fractionDigits* parameter, the number is rounded to the nearest number represented by *fractionDigits* digits. If the number is exactly halfway between two representable numbers, it is rounded away from zero (up if it is positive, down if it is negative). Thus:

0.124.toFixed(2) returns "0.12".
0.125.tofixed(2) returns "0.13", because 0.125 is exactly halfway between 0.12 and 0.13.
0 .126.tofixed(2) returns "0.13".

Given this convention, one might expect 0.045.toFixed(2) to return "0.05", but it returns "0.04". This is because of the way computers represent IEEE 754 floating-point numbers. The IEEE 754 standard uses binary fractions (fractions of 0's and 1's after the dot). Just as some numbers, such as 1/3, are not representable precisely as decimal fractions, other numbers, such as 0.045, are not precisely representable as binary fractions. The IEEE 754 standard dictates that 0.045 be approximated to 0.044999999999999998334665463062265189364552497863769531 25, which is precisely representable as a binary fraction. This approximation is closer to 0.04 than to 0.05, so 0.045.toFixed(2) returns "0.04".

Examples
var num=10.1234
alert("num.toFixed() is " + num.toFixed()) //displays 10
alert("num.toFixed(4) is " + num.toFixed(4)) //displays 10.1234 alert("num.toFixed(2) is " + num.toFixed(2)) //displays 10.12

See also
toExponential, toPrecision, toString

toPrecision

Returns a string representing the Number object to the specified precision.

Method of	Number
Implemented in	JavaScript 1.5
ECMA version	ECMA-262, Edition 3

Syntax
toPrecision([*precision*])

Parameters

precision	An integer specifying the number of digits after the decimal point.

Description
The Number.prototype.toPrecision method returns a string representing a Number object in fixed-point or exponential notation rounded to *precision* significant digits.

If you use the toPrecision method for a numeric literal and the numeric literal has no exponent and no decimal point, leave a space before the dot that precedes the method call to prevent the dot from being interpreted as a decimal point.

If the *precision* argument is omitted, behaves as Number.prototype.toString.

If a number has more digits that requested by the *precision* parameter, the number is rounded to the nearest number represented by *precision* digits. See the discussion of rounding in the description of the toFixed method on page 129, which also applies to toPrecision.

Examples
```
var num=5.123456
alert("num.toPrecision() is " + num.toPrecision()) //displays 5.123456
alert("num.toPrecision(4) is " + num.toPrecision(4)) //displays 5.123
alert("num.toPrecision(2) is " + num.toPrecision(2)) //displays 5.1
alert("num.toPrecision(2) is " + num.toPrecision(1)) //displays 5
alert("num.toPrecision(2) is " + num.toPrecision(1)) //displays 5
alert("1250 .toPrecision() is " + 1250 .toPrecision(2))
//displays 1.3e+3
alert("1250 .toPrecision(5) is " + 1250 .toPrecision(5))
```

```
//displays 1250.0
```

See also
toExponential, toFixed, toString

toSource

Returns a string representing the source code of the object.

Method of	Number
Implemented in	JavaScript 1.3

Syntax
toSource()

Parameters
None

Description
The toSource method returns the following values:

-
- For the built-in Number object, toSource returns the following string indicating that the source code is not available:

```
function Number() {
  [native code]
}
```

- For instances of Number, toSource returns a string representing the source code.

This method is usually called internally by JavaScript and not explicitly in code.

See also
Object.toSource

toString

Returns a string representing the specified Number object.

Method of	Number
Implemented in	JavaScript 1.1
ECMA version	ECMA-262

Syntax
toString()
toString([*radix*])

Parameters

radix	An integer between 2 and 36 specifying the base to use for representing numeric values.

Description

The Number object overrides the toString method of the Object object; it does not inherit Object.toString. For Number objects, the toString method returns a string representation of the object.

JavaScript calls the toString method automatically when a number is to be represented as a text value or when a number is referred to in a string concatenation.

If you use the toString method for a numeric literal and the numeric literal has no exponent and no decimal point, leave a space before the dot that precedes the method call to prevent the dot from being interpreted as a decimal point.

For Number objects and values, the built-in toString method returns the string representing the value of the number.

```
var howMany=10;
alert("howMany.toString() is " + howMany.toString())
alert("45 .toString() is " + 45 .toString())
```

See also
toExponential, toFixed, toPrecision

valueOf

Returns the primitive value of a Number object.

Method of	Number
Implemented in	JavaScript 1.1
ECMA version	ECMA-262

Syntax
valueOf()

Parameters
None

Description
The valueOf method of Number returns the primitive value of a Number object as a number data type.

This method is usually called internally by JavaScript and not explicitly in code.

Examples
```
x = new Number();
alert(x.valueOf())      //displays 0
```

See also
Object.valueOf

Previous **Contents** **Index** **Next**

Last Updated **September 28, 2000**

Object

Object is the primitive JavaScript object type. All JavaScript objects are descended from Object. That is, all JavaScript objects have the methods defined for Object.

Core object	
Implemented in	JavaScript 1.0: toString method. JavaScript 1.1, NES 2.0: added eval and valueOf methods; constructor property. JavaScript 1.2: deprecated eval method. JavaScript 1.3: added toSource method. JavaScript 1.4: removed eval method.
ECMA version	ECMA-262

Created by
The Object constructor:

new Object()

Parameters
None

Property Summary

Property	Description
constructor	Specifies the function that creates an object's prototype.
prototype	Allows the addition of properties to all objects.

Method Summary

Method	Description
eval	Deprecated. Evaluates a string of JavaScript code in the context of the specified object.
toSource	Returns an object literal representing the specified object; you can use this value to create a new object.
toString	Returns a string representing the specified object.

unwatch	Removes a watchpoint from a property of the object.
valueOf	Returns the primitive value of the specified object.
watch	Adds a watchpoint to a property of the object.

constructor

Specifies the function that creates an object's prototype. Note that the value of this property is a reference to the function itself, not a string containing the function's name.

Property of	Object
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description
All objects inherit a constructor property from their prototype:

```
o = new Object // or o = {} in JavaScript 1.2
o.constructor == Object
a = new Array // or a = [] in JavaScript 1.2
a.constructor == Array
n = new Number(3)
n.constructor == Number
```

Even though you cannot construct most HTML objects, you can do comparisons. For example,

```
document.constructor == Document
document.form3.constructor == Form
```

Examples

The following example creates a prototype, Tree, and an object of that type, theTree. The example then displays the constructor property for the object theTree.

```
function Tree(name) {
  this.name=name
}
theTree = new Tree("Redwood")
document.writeln("<B>theTree.constructor is</B> " +
  theTree.constructor + "<P>")
```

This example displays the following output:

```
theTree.constructor is function Tree(name) { this.name = name; }
```

eval

Deprecated. Evaluates a string of JavaScript code in the context of an object.

Method of	Object
Implemented in	JavaScript 1.1, NES 2.0 JavaScript 1.2, NES 3.0: deprecated as method of objects; retained as top-level function. JavaScript 1.4: removed as method of objects.

Syntax
eval(*string*)

Parameters

string	Any string representing a JavaScript expression, statement, or sequence of statements. The expression can include variables and properties of existing objects.

Description
The eval method is no longer available as a method of Object. Use the top-level eval function.

Backward Compatibility

JavaScript 1.2 and 1.3. eval as a method of Object and every object derived from Object is deprecated (but still available).

JavaScript 1.1. eval is a method of Object and every object derived from Object.

See also
eval

prototype

Represents the prototype for this class. You can use the prototype to add properties or methods to all instances of a class. For more information, see Function.prototype.

Property of	Object
Implemented in	JavaScript 1.1
ECMA version	ECMA-262

toSource

Returns a string representing the source code of the object.

Method of	Object
Implemented in	JavaScript 1.3

Syntax
toSource()

Parameters
None

Description
The toSource method returns the following values:

-
- For the built-in Object object, toSource returns the following string indicating that the source code is not available:

```
function Object() {
  [native code]
}
```

- For instances of Object, toSource returns a string representing the source code.
- For custom objects, toSource returns the JavaScript source that defines the object as a string.

This method is usually called internally by JavaScript and not explicitly in code. You can call toSource while debugging to examine the contents of an object.

Examples
The following code defines the Dog object type and creates theDog, an object of type Dog:

```
function Dog(name,breed,color,sex) {
  this.name=name
  this.breed=breed
  this.color=color
  this.sex=sex
}
theDog = new Dog("Gabby","Lab","chocolate","girl")
```

Calling the toSource method of theDog displays the JavaScript source that defines the object:

```
theDog.toSource()
//returns "{name:"Gabby", breed:"Lab", color:"chocolate", sex:"girl"}
```

See also
Object.toString

toString

Returns a string representing the specified object.

Method of	Object
Implemented in	JavaScript 1.0
ECMA version	ECMA-262

Syntax
toString()

Description
Every object has a toString method that is automatically called when it is to be represented as a text value or when an object is referred to in a string concatenation. For example, the following examples require theDog to be represented as a string:

```
document.write(theDog)
document.write("The dog is " + theDog)
```

By default, the toString method is inherited by every object descended from Object. You can override this method for custom objects that you create. If you do not override toString in a custom object, toString returns [object *type*], where *type* is the object type or the name of the constructor function that created the object.

For example:

```
var o = new Object()
o.toString // returns [object Object]
```

Built-in toString methods. Every built-in core JavaScript object overrides the toString method of Object to return an appropriate value. JavaScript calls this method whenever it needs to convert an object to a string.

Overriding the default toString method. You can create a function to be called in place of the default toString method. The toString method takes no arguments and should return a string. The toString method you create can be any value you want, but it will be most useful if it carries information about the object.

The following code defines the Dog object type and creates theDog, an object of type Dog:

```
function Dog(name,breed,color,sex) {
  this.name=name
  this.breed=breed
  this.color=color
  this.sex=sex
}

theDog = new Dog("Gabby","Lab","chocolate","girl")
```

If you call the toString method on this custom object, it returns the default value inherited from Object:

```
theDog.toString() //returns [object Object]
```

The following code creates dogToString, the function that will be used to override the default toString method. This function generates a string containing each property, of the form "property = value;".

```
function dogToString() {
  var ret = "Dog " + this.name + " is [\n"
  for (var prop in this)
    ret += " " + prop + " is " + this[prop] + ";\n"
  return ret + "]"
}
```

The following code assigns the user-defined function to the object's toString method:

```
Dog.prototype.toString = dogToString
```

With the preceding code in place, any time theDog is used in a string context, JavaScript automatically calls the dogToString function, which returns the following string:

```
Dog Gabby is [
  name is Gabby;
  breed is Lab;
  color is chocolate;
  sex is girl;
]
```

An object's toString method is usually invoked by JavaScript, but you can invoke it yourself as follows:

```
var dogString = theDog.toString()
```

Backward Compatibility

JavaScript 1.2. The behavior of the toString method depends on whether you specify LANGUAGE="JavaScript1.2" in the <SCRIPT> tag:

-
- If you specify LANGUAGE="JavaScript1.2" in the <SCRIPT> tag, the toString method returns an object literal.
- If you do not specify LANGUAGE="JavaScript1.2" in the <SCRIPT> tag, the toString method returns [object *type*], as with other JavaScript versions.

Examples

Example 1: The location object. The following example prints the string equivalent of the current location.

```
document.write("location.toString() is " + location.toString() + "<BR>")
```

The output is as follows:

```
location.toString() is file:///C|/TEMP/myprog.htmll
```

Example 2: Object with no string value. Assume you have an Image object named sealife defined as follows:

```
<IMG NAME="sealife" SRC="images\seaotter.gif" ALIGN="left" VSPACE="10">
```

Because the Image object itself has no special toString method, sealife.toString() returns the following:

```
[object Image]
```

Example 3: The radix parameter. The following example prints the string equivalents of the numbers 0 through 9 in decimal and binary.

```
for (x = 0; x < 10; x++) {
  document.write("Decimal: ", x.toString(10), " Binary: ",
    x.toString(2), "<BR>")
}
```

The preceding example produces the following output:

```
Decimal: 0 Binary: 0
Decimal: 1 Binary: 1
Decimal: 2 Binary: 10
Decimal: 3 Binary: 11
Decimal: 4 Binary: 100
Decimal: 5 Binary: 101
Decimal: 6 Binary: 110
Decimal: 7 Binary: 111
Decimal: 8 Binary: 1000
Decimal: 9 Binary: 1001
```

See also
Object.toSource, Object.valueOf

unwatch

Removes a watchpoint set with the watch method.

Method of	Object
Implemented in	JavaScript 1.2, NES 3.0

Syntax
unwatch(*prop*)

Parameters

prop	The name of a property of the object.

Description
The JavaScript debugger has functionality similar to that provided by this method, as well as other debugging options. For information on the debugger, see *Venkman, the new JavaScript Debugger for Netscape 7.x*.

By default, this method is inherited by every object descended from Object.

Example
See watch.

valueOf

Returns the primitive value of the specified object.

Method of	Object
Implemented in	JavaScript 1.1
ECMA version	ECMA-262

Syntax
valueOf()

Parameters
None

Description

JavaScript calls the valueOf method to convert an object to a primitive value. You rarely need to invoke the valueOf method yourself; JavaScript automatically invokes it when encountering an object where a primitive value is expected.

By default, the valueOf method is inherited by every object descended from Object. Every built-in core object overrides this method to return an appropriate value. If an object has no primitive value, valueOf returns the object itself, which is displayed as:

[object Object]

You can use valueOf within your own code to convert a built-in object into a primitive value. When you create a custom object, you can override Object.valueOf to call a custom method instead of the default Object method.

Overriding valueOf for custom objects. You can create a function to be called in place of the default valueOf method. Your function must take no arguments.

Suppose you have an object type myNumberType and you want to create a valueOf method for it. The following code assigns a user-defined function to the object's valueOf method:

myNumberType.prototype.valueOf = new Function(functionText)

With the preceding code in place, any time an object of type myNumberType is used in a context where it is to be represented as a primitive value, JavaScript automatically calls the function defined in the preceding code.

An object's valueOf method is usually invoked by JavaScript, but you can invoke it yourself as follows:

myNumber.valueOf()

Note Objects in string contexts convert via the toString method, which is different from String objects converting to string primitives using valueOf. All string objects have a string conversion, if only "[object *type*]". But many objects do not convert to number, boolean, or function.

See also

parseInt, Object.toString

watch

Watches for a property to be assigned a value and runs a function when that occurs.

Method of	Object
Implemented in	JavaScript 1.2, NES 3.0

Syntax
watch(*prop*, *handler*)

Parameters

prop	The name of a property of the object.
handler	A function to call.

Description
Watches for assignment to a property named prop in this object, calling handler(prop, oldval, newval) whenever prop is set and storing the return value in that property. A watchpoint can filter (or nullify) the value assignment, by returning a modified newval (or oldval).

If you delete a property for which a watchpoint has been set, that watchpoint does not disappear. If you later recreate the property, the watchpoint is still in effect.

To remove a watchpoint, use the unwatch method. By default, the watch method is inherited by every object descended from Object.

The JavaScript debugger has functionality similar to that provided by this method, as well as other debugging options. For information on the debugger, see *Venkman, the new JavaScript Debugger for Netscape 7.x*.

Example

```
<script language="JavaScript1.2">
o = {p:1}
o.watch("p",
  function (id,oldval,newval) {
    document.writeln("o." + id + " changed from "
      + oldval + " to " + newval)
    return newval
  })

o.p = 2
o.p = 3
delete o.p
o.p = 4

o.unwatch('p')
o.p = 5

</script>
```

This script displays the following:

```
o.p changed from 1 to 2
o.p changed from 2 to 3
o.p changed from 3 to 4
```


Last Updated **September 28, 2000**

Packages

A top-level object used to access Java classes from within JavaScript code.

Core object	
Implemented in	JavaScript 1.1, NES 2.0

Created by

The Packages object is a top-level, predefined JavaScript object. You can automatically access it without using a constructor or calling a method.

Description

The Packages object lets you access the public methods and fields of an arbitrary Java class from within JavaScript. The java, netscape, and sun properties represent the packages java.*, netscape.*, and sun.* respectively. Use standard Java dot notation to access the classes, methods, and fields in these packages. For example, you can access a constructor of the Frame class as follows:

```
var theFrame = new Packages.java.awt.Frame();
```

For convenience, JavaScript provides the top-level netscape, sun, and java objects that are synonyms for the Packages properties with the same names. Consequently, you can access Java classes in these packages without the Packages keyword, as follows:

```
var theFrame = new java.awt.Frame();
```

The className property represents the fully qualified path name of any other Java class that is available to JavaScript. You must use the Packages object to access classes

outside the netscape, sun, and java packages.

Property Summary

Property	Description
className	The fully qualified name of a Java class in a package other than netscape, java, or sun that is available to JavaScript.
java	Any class in the Java package java.*.
netscape	Any class in the Java package netscape.*.
sun	Any class in the Java package sun.*.

Examples

The following JavaScript function creates a Java dialog box:

```
function createWindow() {
  var theOwner = new Packages.java.awt.Frame();
  var theWindow = new Packages.java.awt.Dialog(theOwner);
  theWindow.setSize(350,200);
  theWindow.setTitle("Hello, World");
  theWindow.setVisible(true);
}
```

In the previous example, the function instantiates theWindow as a new Packages object.

The setSize, setTitle, and setVisible methods are all available to JavaScript as public methods of java.awt.Dialog.

className

The fully qualified name of a Java class in a package other than netscape, java, or sun that is available to JavaScript.

Property of	Packages
Implemented in	JavaScript 1.1, NES 2.0

Syntax
Packages.*className*

where *classname* is the fully qualified name of a Java class.

Description
You must use the *className* property of the Packages object to access classes outside the netscape, sun, and java packages.

Examples
The following code accesses the constructor of the CorbaObject class in the myCompany package from JavaScript:

```
var theObject = new Packages.myCompany.CorbaObject()
```

In the previous example, the value of the *className* property is myCompany.CorbaObject, the fully qualified path name of the CorbaObject class.

java

Any class in the Java package java.*.

Property of	Packages
Implemented in	JavaScript 1.1, NES 2.0

Syntax
Packages.java

Description
Use the java property to access any class in the java package from within JavaScript. Note that the top-level object java is a synonym for Packages.java.

Examples
The following code accesses the constructor of the java.awt.Frame class:

```
var theOwner = new Packages.java.awt.Frame();
```

You can simplify this code by using the top-level java object to access the constructor as follows:

```
var theOwner = new java.awt.Frame();
```

netscape

Any class in the Java package netscape.*.

Property of	Packages
Implemented in	JavaScript 1.1, NES 2.0

Syntax
Packages.netscape

Description
Use the netscape property to access any class in the netscape package from within JavaScript. Note that the top-level object netscape is a synonym for Packages.netscape.

Examples
See the example for .Packages.java

sun

Any class in the Java package sun.*.

Property of	Packages
Implemented in	JavaScript 1.1, NES 2.0

Syntax
Packages.sun

Description
Use the sun property to access any class in the sun package from within JavaScript. Note that the top-level object sun is a synonym for Packages.sun.

Examples
See the example for Packages.java

Last Updated **September 28, 2000**

RegExp

A regular expression object contains the pattern of a regular expression. It has properties and methods for using that regular expression to find and replace matches in strings.

In addition to the properties of an individual regular expression object that you create using the RegExp constructor function, the predefined RegExp object has static properties that are set whenever any regular expression is used.

Core object	
Implemented in	JavaScript 1.2, NES 3.0 JavaScript 1.3: added toSource method. JavaScript 1.5, NES 6.0: added m flag, non-greedy modifier, non-capturing parentheses, lookahead assertions. ECMA 262, Edition 3

Created by

A literal text format or the RegExp constructor function.

The literal format is used as follows:

/pattern/flags

The constructor function is used as follows:

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

Parameters

pattern	The text of the regular expression.
flags	If specified, flags can have any combination of the following values: • • g: global match • i: ignore case • m: match over multiple lines

Notice that the parameters to the literal format do not use quotation marks to indicate strings, while the parameters to the constructor function do use quotation marks. So the following expressions create the same regular expression:

/ab+c/i
new RegExp("ab+c", "i")

Description
When using the constructor function, the normal string escape rules (preceding special characters with \ when included in a string) are necessary. For example, the following are equivalent:

re = new RegExp("\\w+")
re = /\w+/

The following table provides a complete list and description of the special characters that can be used in regular expressions.

Table 1.1 Special characters in regular expressions.

Character	Meaning
\	For characters that are usually treated literally, indicates that the next character is special and not to be interpreted literally. For example, /b/ matches the character 'b'. By placing a backslash in front of b, that is by using /\b/, the character becomes special to mean match a word boundary. -or- For characters that are usually treated specially, indicates that the next character is not special and should be interpreted literally. For example, * is a special character that means 0 or more occurrences of the preceding character should be matched; for example, /a*/ means match 0 or more a's. To match * literally, precede the it with a backslash; for example, /a*/ matches 'a*'.
^	Matches beginning of input. If the multiline flag is set to true, also matches immediately after a line break character. For example, /^A/ does not match the 'A' in "an A", but does match the first 'A' in "An A."
$	Matches end of input. If the multiline flag is set to true, also matches immediately before a line break character. For example, /t$/ does not match the 't' in "eater", but does match it in "eat".
*	Matches the preceding item 0 or more times. For example, /bo*/ matches 'boooo' in "A ghost booooed" and 'b' in "A bird warbled", but nothing in "A goat grunted".

+	Matches the preceding item 1 or more times. Equivalent to {1,}. For example, /a+/ matches the 'a' in "candy" and all the a's in "caaaaaaandy".
?	Matches the preceding item 0 or 1 time. For example, /e?le?/ matches the 'el' in "angel" and the 'le' in "angle." If used immediately after any of the quantifiers *, +, ?, or {}, makes the quantifier non-greedy (matching the minimum number of times), as opposed to the default, which is greedy (matching the maximum number of times). Also used in lookahead assertions, described under (?=), (?!), and (?:) in this table.
.	(The decimal point) matches any single character except the newline character. For example, /.n/ matches 'an' and 'on' in "nay, an apple is on the tree", but not 'nay'.
(x)	Matches 'x' and remembers the match. These are called capturing parentheses. For example, /(foo)/ matches and remembers 'foo' in "foo bar." The matched substring can be recalled from the resulting array's elements [1], ..., [n] or from the predefined RegExp object's properties $1, ..., $9.
(?:x)	Matches 'x' but does not remember the match. These are called non-capturing parentheses. The matched substring can not be recalled from the resulting array's elements [1], ..., [n] or from the predefined RegExp object's properties $1, ..., $9.
x(?=y)	Matches 'x' only if 'x' is followed by 'y'. For example, /Jack(?=Sprat)/ matches 'Jack' only if it is followed by 'Sprat'. /Jack(?=Sprat|Frost)/matches 'Jack' only if it is followed by 'Sprat' or 'Frost'. However, neither 'Sprat' nor 'Frost' is part of the match results.

x(?!y)	Matches 'x' only if 'x' is not followed by 'y'. For example, /\d+(?!\.)/ matches a number only if it is not followed by a decimal point. /\d+(?!\.)/.exec("3.141") matches 141 but not 3.141.
x|y	Matches either 'x' or 'y'. For example, /green|red/ matches 'green' in "green apple" and 'red' in "red apple."
{n}	Where n is a positive integer. Matches exactly n occurrences of the preceding item. For example, /a{2}/ doesn't match the 'a' in "candy," but it matches all of the a's in "caandy," and the first two a's in "caaandy."
{n,}	Where n is a positive integer. Matches at least n occurrences of the preceding item. For example, /a{2,} doesn't match the 'a' in "candy", but matches all of the a's in "caandy" and in "caaaaaaandy."
{n,m}	Where n and m are positive integers. Matches at least n and at most m occurrences of the preceding item. For example, /a{1,3}/ matches nothing in "cndy", the 'a' in "candy," the first two a's in "caandy," and the first three a's in "caaaaaaandy". Notice that when matching "caaaaaaandy", the match is "aaa", even though the original string had more a's in it.
[xyz]	A character set. Matches any one of the enclosed characters. You can specify a range of characters by using a hyphen. For example, [abcd] is the same as [a-c]. They match the 'b' in "brisket" and the 'c' in "ache".

[^xyz]	A negated or complemented character set. That is, it matches anything that is not enclosed in the brackets. You can specify a range of characters by using a hyphen. For example, [^abc] is the same as [^a-c]. They initially match 'r' in "brisket" and 'h' in "chop."
[\b]	Matches a backspace. (Not to be confused with \b.)
\b	Matches a word boundary, such as a space. (Not to be confused with [\b].) For example, /\bn\w/ matches the 'no' in "noonday";/\wy\b/ matches the 'ly' in "possibly yesterday."
\B	Matches a non-word boundary. For example, /\w\Bn/ matches 'on' in "noonday", and /y\B\w/ matches 'ye' in "possibly yesterday."
\c*X*	Where *X* is a letter from A - Z. Matches a control character in a string. For example, /\cM/ matches control-M in a string.
\d	Matches a digit character. Equivalent to [0-9]. For example, /\d/ or /[0-9]/ matches '2' in "B2 is the suite number."
\D	Matches any non-digit character. Equivalent to [^0-9]. For example, /\D/ or /[^0-9]/ matches 'B' in "B2 is the suite number."
\f	Matches a form-feed.

\n	Matches a linefeed.
\r	Matches a carriage return.
\s	Matches a single white space character, including space, tab, form feed, line feed. Equivalent to [\f\n\r\t\u00A0\u2028\u2029]. For example, /\s\w*/ matches ' bar' in "foo bar."
\S	Matches a single character other than white space. Equivalent to [^ \f\n\r\t\u00A0\u2028\u2029]. For example, /\S/\w* matches 'foo' in "foo bar."
\t	Matches a tab.
\v	Matches a vertical tab.
\w	Matches any alphanumeric character including the underscore. Equivalent to [A-Za-z0-9_]. For example, /\w/ matches 'a' in "apple," '5' in "$5.28," and '3' in "3D."
\W	Matches any non-word character. Equivalent to [^A-Za-z0-9_]. For example, /\W/ or /[^$A-Za-z0-9_]/ matches '%' in "50%."

\n	Where *n* is a positive integer. A back reference to the last substring matching the *n* parenthetical in the regular expression (counting left parentheses). For example, /apple(,)\sorange\1/ matches 'apple, orange', in "apple, orange, cherry, peach." A more complete example follows this table.
\0	Matches a NUL character. Do not follow this with another digit.
\xhh	Matches the character with the code hh (two hexadecimal digits)
\uhhhh	Matches the character with code hhhh (four hexadecimal digits).

The literal notation provides compilation of the regular expression when the expression is evaluated. Use literal notation when the regular expression will remain constant. For example, if you use literal notation to construct a regular expression used in a loop, the regular expression won't be recompiled on each iteration.

The constructor of the regular expression object, for example, new RegExp("ab+c"), provides runtime compilation of the regular expression. Use the constructor function when you know the regular expression pattern will be changing, or you don't know the pattern and are getting it from another source, such as user input.

A separate predefined RegExp object is available in each window; that is, each separate thread of JavaScript execution gets its own RegExp object. Because each script runs to completion without interruption in a thread, this assures that different scripts do not overwrite values of the RegExp object.

Property Summary

Note that several of the RegExp properties have both long and short (Perl-like) names. Both names always refer to the same value. Perl is the programming language from which JavaScript modeled its regular expressions.

Property	Description
constructor	Specifies the function that creates an object's prototype.
global	Whether to test the regular expression against all possible matches in a string, or only against the first. As of JavaScript 1.5, a property of a RegExp instance, not the RegExp object.
ignoreCase	Whether to ignore case while attempting a match in a string. As of JavaScript 1.5, a property of a RegExp instance, not the RegExp object.
lastIndex	The index at which to start the next match. As of JavaScript 1.5, a property of a RegExp instance, not the RegExp object.
multiline	Whether or not to search in strings across multiple lines. As of JavaScript 1.5, a property of a RegExp instance, not the RegExp object.
prototype	Allows the addition of properties to all objects.
source	The text of the pattern. As of JavaScript 1.5, a property of a RegExp instance, not the RegExp object.

Method Summary

Method	Description
exec	Executes a search for a match in its string parameter.
test	Tests for a match in its string parameter.
toSource	Returns an object literal representing the specified object; you can use this value to create a new object. Overrides the Object.toSource method.
toString	Returns a string representing the specified object. Overrides the Object.toString method.

In addition, this object inherits the watch and unwatch methods from Object.

Examples

Example 1. The following script uses the replace method to switch the words in the string. In the replacement text, the script uses "$1" and "$2" to indicate the results of the corresponding matching parentheses in the regular expression pattern.

```
<SCRIPT>
re = /(\w+)\s(\w+)/;
str = "John Smith";
newstr=str.replace(re, "$2, $1");
document.write(newstr)
</SCRIPT>
```

This displays "Smith, John".

Example 2. In the following example, RegExp.input is set by the Change event. In the getInfo function, the exec method uses the value of RegExp.input as its argument.

```
<HTML>

<SCRIPT>
function getInfo() {
  re = /(\w+)\s(\d+)/;
  var m = re.exec();
  window.alert(m[] + ", your age is " + m[2]);
}
</SCRIPT>

Enter your first name and your age, and then press Enter.

<FORM>
<INPUT TYPE:"TEXT" NAME="NameAge" onChange="getInfo(this);">
</FORM>

</HTML>
```

constructor

Specifies the function that creates an object's prototype. Note that the value of this property is a reference to the function itself, not a string containing the function's name.

Property of	RegExp
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description
See Object.constructor.

exec

Executes the search for a match in a specified string. Returns a result array.

Method of	RegExp
Implemented in	JavaScript 1.2, NES 3.0
ECMA version	ECMA 262, Edition 3 (first syntax only)

Syntax

regexp.exec([*str*])
regexp([*str*])

Parameters

regexp	The name of the regular expression. It can be a variable name or a literal.
str	The string against which to match the regular expression.

Description

As shown in the syntax description, a regular expression's exec method can be called either directly, (with regexp.exec(str)) or indirectly (with regexp(str)).

If you are executing a match simply to find true or false, use the test method or the String search method.

If the match succeeds, the exec method returns an array and updates properties of the regular expression object. If the match fails, the exec method returns null.

Consider the following example:

```
<SCRIPT LANGUAGE="JavaScript1.2">
//Match one d followed by one or more b's followed by one d
//Remember matched b's and the following d
//Ignore case
myRe=/d(b+)(d)/ig;
myArray = myRe.exec("cdbBdbsbz");
</SCRIPT>
```

The following table shows the results for this script:

Object	Property/Index	Description	Example
myArray		The contents of myArray.	["dbBd", "bB", "d"]
	index	The 0-based index of the match in the string.	1
	input	The original string.	cdbBdbsbz

	[0]	The last matched characters.	dbBd
	[1], ...[*n*]	The parenthesized substring matches, if any. The number of possible parenthesized substrings is unlimited.	[1] = bB [2] = d
myRe	lastIndex	The index at which to start the next match.	5
	ignoreCase	Indicates if the "i" flag was used to ignore case.	true
	global	Indicates if the "g" flag was used for a global match.	true
	multiline	Indicates if the "m" flag was used for a global match.	false
	source	The text of the pattern.	d(b+)(d)

If your regular expression uses the "g" flag, you can use the exec method multiple times to find successive matches in the same string. When you do so, the search starts at the substring of str specified by the regular expression's lastIndex property. For example, assume you have this script:

```
<SCRIPT LANGUAGE="JavaScript1.2">
myRe=/ab*/g;
```

```
str = "abbcdefabh";
myArray = myRe.exec(str);
document.writeln("Found " + myArray[0] +
  ". Next match starts at " + myRe.lastIndex)
mySecondArray = myRe.exec(str);
document.writeln("Found " + mySecondArray[0] +
  ". Next match starts at " + myRe.lastIndex)
</SCRIPT>
```

This script displays the following text:

```
Found abb. Next match starts at 3
Found ab. Next match starts at 9
```

Examples

In the following example, the user enters a name and the script executes a match against the input. It then cycles through the array to see if other names match the user's name.

This script assumes that first names of registered party attendees are preloaded into the array A, perhaps by gathering them from a party database.

```
<HTML>

<SCRIPT LANGUAGE="JavaScript1.2">
A = ["Frank", "Emily", "Jane", "Harry", "Nick", "Beth", "Rick",
     "Terrence", "Carol", "Ann", "Terry", "Frank", "Alice", "Rick",
     "Bill", "Tom", "Fiona", "Jane", "William", "Joan", "Beth"]

function lookup() {
  firstName = /\w+/i();
  if (!firstName)
    window.alert (RegExp.input + " isn't a name!");
  else {
    count = 0;
    for (i=0; i<A.length; i++)
      if (firstName[0].toLowerCase() == A[i].toLowerCase()) count++;
    if (count ==1)
      midstring = " other has ";
    else
      midstring = " others have ";
    window.alert ("Thanks, " + count + midstring + "the same name!")
  }
}
```

</SCRIPT>

Enter your first name and then press Enter.

<FORM> <INPUT TYPE:"TEXT" NAME="FirstName" onChange="lookup(this);"> </FORM>

</HTML>

global

Whether or not the "g" flag is used with the regular expression.

Property of	RegExp instances
Read-only	
Implemented in	JavaScript 1.2, NES 3.0
ECMA version	ECMA 262, Edition 3

Description
global is a property of an individual regular expression object.

The value of global is true if the "g" flag was used; otherwise, false. The "g" flag indicates that the regular expression should be tested against all possible matches in a string.

You cannot change this property directly.

ignoreCase

Whether or not the "i" flag is used with the regular expression.

Property of	RegExp instances
Read-only	
Implemented in	JavaScript 1.2, NES 3.0
ECMA version	ECMA 262, Edition 3

Description
ignoreCase is a property of an individual regular expression object.

The value of ignoreCase is true if the "i" flag was used; otherwise, false. The "i" flag indicates that case should be ignored while attempting a match in a string.

You cannot change this property directly.

lastIndex

A read/write integer property that specifies the index at which to start the next match.

Property of	RegExp instances
Implemented in	JavaScript 1.2, NES 3.0

ECMA version	ECMA 262, Edition 3

Description
lastIndex is a property of an individual regular expression object.

This property is set only if the regular expression used the "g" flag to indicate a global search. The following rules apply:

-
- If lastIndex is greater than the length of the string, regexp.test and regexp.exec fail, and lastIndex is set to 0.
- If lastIndex is equal to the length of the string and if the regular expression matches the empty string, then the regular expression matches input starting at lastIndex .
- If lastIndex is equal to the length of the string and if the regular expression does not match the empty string, then the regular expression mismatches input, and lastIndex is reset to 0.
- Otherwise, lastIndex is set to the next position following the most recent match.

For example, consider the following sequence of statements:

re = /(hi)?/g	Matches the empty string.
re("hi")	Returns ["hi", "hi"] with lastIndex equal to 2.

re("hi")	Returns [""], an empty array whose zeroth element is the match string. In this case, the empty string because lastIndex was 2 (and still is 2) and "hi" has length 2.

multiline

Reflects whether or not to search in strings across multiple lines.

Property of	RegExp instances
Static	
Implemented in	JavaScript 1.2, NES 3.0
ECMA version	ECMA 262, Edition 3

Description
multiline is a property of an individual regular expression object..

The value of multiline is true if the "m" flag was used; otherwise, false. The "m" flag indicates that a multiline input string should be treated as multiple lines. For example, if "m" is used, "^" and "$" change from matching at only the start or end of the entire string to the start or end of any line within the string.

You cannot change this property directly.

prototype

Represents the prototype for this class. You can use the prototype to add properties or

methods to all instances of a class. For information on prototypes, see Function.prototype.

Property of	RegExp
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

source

A read-only property that contains the text of the pattern, excluding the forward slashes.

Property of	RegExp instances
Read-only	
Implemented in	JavaScript 1.2, NES 3.0
ECMA version	ECMA 262, Edition 3

Description
source is a property of an individual regular expression object.

You cannot change this property directly.

test

Executes the search for a match between a regular expression and a specified string. Returns true or false.

Method of	RegExp
Implemented in	JavaScript 1.2, NES 3.0
ECMA version	ECMA 262, Edition 3

Syntax
regexp.test([*str*])

Parameters

regexp	The name of the regular expression. It can be a variable name or a literal.
str	The string against which to match the regular expression.

Description
When you want to know whether a pattern is found in a string use the test method (similar to the String.search method); for more information (but slower execution) use

the exec method (similar to the String.match method).

Example
The following example prints a message which depends on the success of the test:

```
function testinput(re, str){
  if (re.test(str))
    midstring = " contains ";
  else
    midstring = " does not contain ";
  document.write (str + midstring + re.source);
}
```

toSource

Returns a string representing the source code of the object.

Method of	RegExp
Implemented in	JavaScript 1.3

Syntax
toSource()

Parameters
None

Description
The toSource method returns the following values:

-
- For the built-in RegExp object, toSource returns the following string indicating

that the source code is not available:

```
function Boolean() {
  [native code]
}
```

- For instances of RegExp, toSource returns a string representing the source code.

This method is usually called internally by JavaScript and not explicitly in code.

See also
Object.toSource

toString

Returns a string representing the specified object.

Method of	RegExp
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA 262, Edition 3

Syntax
toString()

Parameters
None.

Description
The RegExp object overrides the toString method of the Object object; it does not

inherit Object.toString. For RegExp objects, the toString method returns a string representation of the object.

Examples

The following example displays the string value of a RegExp object:

```
myExp = new RegExp("a+b+c");
alert(myExp.toString())        displays "/a+b+c/"
```

See also

Object.toString

Previous **Contents** **Index** **Next**

Last Updated **September 28, 2000**

String

An object representing a series of characters in a string.

Core object	
Implemented in	JavaScript 1.0: Create a String object only by quoting characters. JavaScript 1.1, NES 2.0: added String constructor; added prototype property; added split method; added ability to pass strings among scripts in different windows or frames (in previous releases, you had to add an empty string to another window's string to refer to it). JavaScript 1.2, NES 3.0: added concat, match, replace, search, slice, and substr methods. JavaScript 1.3: added toSource method; changed charCodeAt, fromCharCode, and replace methods.
ECMA version	ECMA-262

Created by

The String constructor:

new String(*string*)

Parameters

string	Any string.

Description

The String object is a wrapper around the string primitive data type. Do not confuse a string literal with the String object. For example, the following code creates the string literal s1 and also the String object s2:

```
s1 = "foo" // creates a string literal value
s2 = new String("foo") // creates a String object
```

You can call any of the methods of the String object on a string literal value-JavaScript automatically converts the string literal to a temporary String object, calls the method, then discards the temporary String object. You can also use the String.length property with a string literal.

You should use string literals unless you specifically need to use a String object, because String objects can have counterintuitive behavior. For example:

```
s1 = "2 + 2" // creates a string literal value
s2 = new String("2 + 2") // creates a String object
eval(s1)    // returns the number 4
eval(s2)    // returns the string "2 + 2"
```

A string can be represented as a literal enclosed by single or double quotation marks; for example, "Netscape" or `Netscape'.

You can convert the value of any object into a string using the top-level String function.

Property Summary

Property	Description
constructor	Specifies the function that creates an object's prototype.
length	Reflects the length of the string.
prototype	Allows the addition of properties to a String object.

Method Summary

Method	Description
anchor	Creates an HTML anchor that is used as a hypertext target.
big	Causes a string to be displayed in a big font as if it were in a BIG tag.
blink	Causes a string to blink as if it were in a BLINK tag.

bold	Causes a string to be displayed as if it were in a B tag.
charAt	Returns the character at the specified index.
charCodeAt	Returns a number indicating the Unicode value of the character at the given index.
concat	Combines the text of two strings and returns a new string.
fixed	Causes a string to be displayed in fixed-pitch font as if it were in a TT tag.
fontcolor	Causes a string to be displayed in the specified color as if it were in a <FONT COLOR=color> tag.
fontsize	Causes a string to be displayed in the specified font size as if it were in a <FONT SIZE=size> tag.
fromCharCode	Returns a string created by using the specified sequence of Unicode values. This is a method on the String class, not on a String instance.

indexOf	Returns the index within the calling String object of the first occurrence of the specified value, or -1 if not found.
italics	Causes a string to be italic, as if it were in an I tag.
lastIndexOf	Returns the index within the calling String object of the last occurrence of the specified value, or -1 if not found.
link	Creates an HTML hypertext link that requests another URL.
match	Used to match a regular expression against a string.
replace	Used to find a match between a regular expression and a string, and to replace the matched substring with a new substring.
search	Executes the search for a match between a regular expression and a specified string.
slice	Extracts a section of a string and returns a new string.

small	Causes a string to be displayed in a small font, as if it were in a SMALL tag.
split	Splits a String object into an array of strings by separating the string into substrings.
strike	Causes a string to be displayed as struck-out text, as if it were in a STRIKE tag.
sub	Causes a string to be displayed as a subscript, as if it were in a SUB tag.
substr	Returns the characters in a string beginning at the specified location through the specified number of characters.
substring	Returns the characters in a string between two indexes into the string.
sup	Causes a string to be displayed as a superscript, as if it were in a SUP tag.
toLowerCase	Returns the calling string value converted to lowercase.

toSource	Returns an object literal representing the specified object; you can use this value to create a new object. Overrides the Object.toSource method.
toString	Returns a string representing the specified object. Overrides the Object.toString method.
toUpperCase	Returns the calling string value converted to uppercase.
valueOf	Returns the primitive value of the specified object. Overrides the Object.valueOf method.

In addition, this object inherits the watch and unwatch methods from Object.

Examples
Example 1: String literal. The following statement creates a string literal:

```
var last_name = "Schaefer"
```

Example 2: String literal properties. The following statements evaluate to 8, "SCHAEFER," and "schaefer":

```
last_name.length
last_name.toUpperCase()
last_name.toLowerCase()
```

Example 3: Accessing individual characters in a string. You can think of a string as an array of characters. In this way, you can access the individual characters in the string by indexing that array. For example, the following code displays "The first character in the string is H":

```
var myString = "Hello"
myString[0] // returns "H"
```

Example 4: Pass a string among scripts in different windows or frames. The following code creates two string variables and opens a second window:

```
var lastName = "Schaefer"
var firstName = "Jesse"
empWindow=window.open('string2.html','window1','width=300,height=300')
```

If the HTML source for the second window (string2.html) creates two string variables, empLastName and empFirstName, the following code in the first window assigns values to the second window's variables:

```
empWindow.empFirstName=firstName
empWindow.empLastName=lastName
```

The following code in the first window displays the values of the second window's variables:

```
alert('empFirstName in empWindow is ' + empWindow.empFirstName)
alert('empLastName in empWindow is ' + empWindow.empLastName)
```

anchor

Creates an HTML anchor that is used as a hypertext target.

Method of	String
Implemented in	JavaScript 1.0, NES 2.0

Syntax

anchor(*nameAttribute*)

Parameters

nameAttribute	A string.

Description
Use the anchor method with the document.write or document.writeln methods to programmatically create and display an anchor in a document. Create the anchor with the anchor method, and then call write or writeln to display the anchor in a document. In server-side JavaScript, use the write function to display the anchor.

In the syntax, the text string represents the literal text that you want the user to see. The nameAttribute string represents the NAME attribute of the A tag.

Anchors created with the anchor method become elements in the document.anchors array.

Examples
The following example opens the msgWindow window and creates an anchor for the table of contents:

```
var myString="Table of Contents"
msgWindow.document.writeln(myString.anchor("contents_anchor"))
```

The previous example produces the same output as the following HTML:

```
<A NAME="contents_anchor">Table of Contents</A>
```

See also
String.link

big

Causes a string to be displayed in a big font as if it were in a BIG tag.

Method of	String
Implemented in	JavaScript 1.0, NES 2.0

Syntax
big()

Parameters
None

Description
Use the big method with the write or writeln methods to format and display a string in a document. In server-side JavaScript, use the write function to display the string.

Examples
The following example uses string methods to change the size of a string:

```
var worldString="Hello, world"

document.write(worldString.small())
document.write("<P>" + worldString.big())
document.write("<P>" + worldString.fontsize(7))
```

The previous example produces the same output as the following HTML:

```
<SMALL>Hello, world</SMALL>
<P><BIG>Hello, world</BIG>
<P><FONTSIZE=7>Hello, world</FONTSIZE>
```

See also
String.fontsize, String.small

blink

Causes a string to blink as if it were in a BLINK tag.

Method of	String
Implemented in	JavaScript 1.0, NES 2.0

Syntax
blink()

Parameters
None

Description
Use the blink method with the write or writeln methods to format and display a string in a document. In server-side JavaScript, use the write function to display the string.

Examples
The following example uses string methods to change the formatting of a string:

```
var worldString="Hello, world"

document.write(worldString.blink())
document.write("<P>" + worldString.bold())
document.write("<P>" + worldString.italics())
document.write("<P>" + worldString.strike())
```

The previous example produces the same output as the following HTML:

```
<BLINK>Hello, world</BLINK>
<P><B>Hello, world</B>
<P><I>Hello, world</I>
<P><STRIKE>Hello, world</STRIKE>
```

See also
String.bold, String.italics, String.strike

bold

Causes a string to be displayed as bold as if it were in a B tag.

Method of	String
Implemented in	JavaScript 1.0, NES 2.0

Syntax
bold()

Parameters
None

Description
Use the bold method with the write or writeln methods to format and display a string in a document. In server-side JavaScript, use the write function to display the string.

Examples
The following example uses string methods to change the formatting of a string:

```
var worldString="Hello, world"
document.write(worldString.blink())
document.write("<P>" + worldString.bold())
document.write("<P>" + worldString.italics())
document.write("<P>" + worldString.strike())
```

The previous example produces the same output as the following HTML:

```
<BLINK>Hello, world</BLINK>
<P><B>Hello, world</B>
<P><I>Hello, world</I>
<P><STRIKE>Hello, world</STRIKE>
```

See also
String.blink, String.italics, String.strike

charAt

Returns the specified character from the string.

Method of	String
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
charAt(*index*)

Parameters

index	An integer between 0 and 1 less than the length of the string.

Description
Characters in a string are indexed from left to right. The index of the first character is 0, and the index of the last character in a string called stringName is stringName.length - 1. If the index you supply is out of range, JavaScript returns an empty string.

Examples
The following example displays characters at different locations in the string "Brave new world":

```
var anyString="Brave new world"

document.writeln("The character at index 0 is " + anyString.charAt(0))
document.writeln("The character at index 1 is " + anyString.charAt(1))
document.writeln("The character at index 2 is " + anyString.charAt(2))
document.writeln("The character at index 3 is " + anyString.charAt(3))
document.writeln("The character at index 4 is " + anyString.charAt(4))
```

These lines display the following:

```
The character at index 0 is B
The character at index 1 is r
The character at index 2 is a
The character at index 3 is v
The character at index 4 is e
```

See also
String.indexOf, String.lastIndexOf, String.split

charCodeAt

Returns a number indicating the Unicode value of the character at the given index.

Method of	String

Implemented in	JavaScript 1.2, NES 3.0 JavaScript 1.3: returns a Unicode value rather than an ISO-Latin-1 value.
ECMA version	ECMA-262

Syntax
charCodeAt([*index*])

Parameters

index	An integer between 0 and 1 less than the length of the string. The default value is 0.

Description
Unicode values range from 0 to 65,535. The first 128 Unicode values are a direct match of the ASCII character set. For information on Unicode, see the *Core JavaScript Guide*.

Backward Compatibility

JavaScript 1.2. The charCodeAt method returns a number indicating the ISO-Latin-1 codeset value of the character at the given index. The ISO-Latin-1 codeset ranges from 0 to 255. The first 0 to 127 are a direct match of the ASCII character set.

Example
The following example returns 65, the Unicode value for A.

```
"ABC".charCodeAt(0) // returns 65
```

concat

Combines the text of two or more strings and returns a new string.

Method of	String
Implemented in	JavaScript 1.2, NES 3.0

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

Parameters

string2... string*N*	Strings to concatenate to this string.

Description
concat combines the text from one or more strings and returns a new string. Changes to the text in one string do not affect the other string.

Example
The following example combines two strings into a new string.

```
s1="Oh "
s2="what a beautiful "
```

```
s3="mornin'."
s4=s1.concat(s2,s3) // returns "Oh what a beautiful mornin'."
```

constructor

Specifies the function that creates an object's prototype. Note that the value of this property is a reference to the function itself, not a string containing the function's name.

Property of	String
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Description
See Object.constructor.

fixed

Causes a string to be displayed in fixed-pitch font as if it were in a TT tag.

Method of	String
Implemented in	JavaScript 1.0, NES 2.0

Syntax

fixed()

Parameters
None

Description
Use the fixed method with the write or writeln methods to format and display a string in a document. In server-side JavaScript, use the write function to display the string.

Examples
The following example uses the fixed method to change the formatting of a string:

```
var worldString="Hello, world"
document.write(worldString.fixed())
```

The previous example produces the same output as the following HTML:

```
<TT>Hello, world</TT>
```

fontcolor

Causes a string to be displayed in the specified color as if it were in a <FONT COLOR=color> tag.

Method of	String
Implemented in	JavaScript 1.0, NES 2.0

Syntax
fontcolor(*color*)

Parameters

color	A string expressing the color as a hexadecimal RGB triplet or as a string literal. String literals for color names are listed in the *Core JavaScript Guide*.

Description

Use the fontcolor method with the write or writeln methods to format and display a string in a document. In server-side JavaScript, use the write function to display the string.

If you express color as a hexadecimal RGB triplet, you must use the format rrggbb. For example, the hexadecimal RGB values for salmon are red=FA, green=80, and blue=72, so the RGB triplet for salmon is "FA8072".

The fontcolor method overrides a value set in the fgColor property.

Examples

The following example uses the fontcolor method to change the color of a string:

```
var worldString="Hello, world"

document.write(worldString.fontcolor("maroon") +
  " is maroon in this line")
document.write("<P>" + worldString.fontcolor("salmon") +
  " is salmon in this line")
document.write("<P>" + worldString.fontcolor("red") +
  " is red in this line")

document.write("<P>" + worldString.fontcolor("8000") +
  " is maroon in hexadecimal in this line")
document.write("<P>" + worldString.fontcolor("FA8072") +
  " is salmon in hexadecimal in this line")
document.write("<P>" + worldString.fontcolor("FF00") +
  " is red in hexadecimal in this line")
```

The previous example produces the same output as the following HTML:

<FONT COLOR="maroon">Hello, world</FONT> is maroon in this line
<P><FONT COLOR="salmon">Hello, world</FONT> is salmon in this line
<P><FONT COLOR="red">Hello, world</FONT> is red in this line

<FONT COLOR="8000">Hello, world</FONT>
is maroon in hexadecimal in this line
<P><FONT COLOR="FA8072">Hello, world</FONT>
is salmon in hexadecimal in this line
<P><FONT COLOR="FF00">Hello, world</FONT>
is red in hexadecimal in this line

fontsize

Causes a string to be displayed in the specified font size as if it were in a <FONT SIZE=size> tag.

Method of	String
Implemented in	JavaScript 1.0, NES 2.0

Syntax
fontsize(*size*)

Parameters

size	An integer between 1 and 7, a string representing a signed integer between 1 and 7.

Description
Use the fontsize method with the write or writeln methods to format and display a string in a document. In server-side JavaScript, use the write function to display the string.

When you specify size as an integer, you set the size of stringName to one of the 7 defined sizes. When you specify size as a string such as "-2", you adjust the font size of stringName relative to the size set in the BASEFONT tag.

Examples
The following example uses string methods to change the size of a string:

```
var worldString="Hello, world"

document.write(worldString.small())
document.write("<P>" + worldString.big())
document.write("<P>" + worldString.fontsize(7))
```

The previous example produces the same output as the following HTML:

```
<SMALL>Hello, world</SMALL>
<P><BIG>Hello, world</BIG>
<P><FONTSIZE=7>Hello, world</FONTSIZE>
```

See also
String.big, String.small

fromCharCode

Returns a string created by using the specified sequence of Unicode values.

Method of	String
Static	
Implemented in	JavaScript 1.2, NES 3.0 JavaScript 1.3: uses a Unicode value rather than an ISO-Latin-1 value.
ECMA version	ECMA-262

Syntax
fromCharCode(*num1*, ..., *numN*)

Parameters

num1, ..., num*N*	A sequence of numbers that are Unicode values.

Description
This method returns a string and not a String object.

Because fromCharCode is a static method of String, you always use it as String.fromCharCode(), rather than as a method of a String object you created.

Backward Compatibility

JavaScript 1.2. The fromCharCode method returns a string created by using the specified sequence of ISO-Latin-1 codeset values.

Examples
The following example returns the string "ABC".

String.fromCharCode(65,66,67)

indexOf

Returns the index within the calling String object of the first occurrence of the specified value, starting the search at fromIndex, or -1 if the value is not found.

Method of	String
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
indexOf(*searchValue*[, *fromIndex*])

Parameters

searchValue	A string representing the value to search for.

fromIndex	The location within the calling string to start the search from. It can be any integer between 0 and the length of the string. The default value is 0.

Description

Characters in a string are indexed from left to right. The index of the first character is 0, and the index of the last character of a string called stringName is stringName.length - 1.

```
"Blue Whale".indexOf("Blue")    // returns 0
"Blue Whale".indexOf("Blute")   // returns -1
"Blue Whale".indexOf("Whale",0) // returns 5
"Blue Whale".indexOf("Whale",5) // returns 5
"Blue Whale".indexOf("",9)      // returns 9
"Blue Whale".indexOf("",10)     // returns 10
"Blue Whale".indexOf("",11)     // returns 10
```

The indexOf method is case sensitive. For example, the following expression returns -1:

```
"Blue Whale".indexOf("blue")
```

Examples

Example 1. The following example uses indexOf and lastIndexOf to locate values in the string "Brave new world."

```
var anyString="Brave new world"

// Displays 8
document.write("<P>The index of the first w from the beginning is " +
  anyString.indexOf("w"))
// Displays 10
document.write("<P>The index of the first w from the end is " +
  anyString.lastIndexOf("w"))
// Displays 6
document.write("<P>The index of 'new' from the beginning is " +
  anyString.indexOf("new"))
// Displays 6
document.write("<P>The index of 'new' from the end is " +
  anyString.lastIndexOf("new"))
```

Example 2. The following example defines two string variables. The variables contain the same string except that the second string contains uppercase letters. The first writeln method displays 19. But because the indexOf method is case sensitive, the string "cheddar" is not found in myCapString, so the second writeln method displays -1.

```
myString="brie, pepper jack, cheddar"
myCapString="Brie, Pepper Jack, Cheddar"
document.writeln('myString.indexOf("cheddar") is ' +
  myString.indexOf("cheddar"))
document.writeln('<P>myCapString.indexOf("cheddar") is ' +
  myCapString.indexOf("cheddar"))
```

Example 3. The following example sets count to the number of occurrences of the letter x in the string str:

```
count = 0;
pos = str.indexOf("x");
while ( pos != -1 ) {
  count++;
  pos = str.indexOf("x",pos+1);
}
```

See also
String.charAt, String.lastIndexOf, String.split

italics

Causes a string to be italic, as if it were in an <I> tag.

Method of	String
Implemented in	JavaScript 1.0, NES 2.0

Syntax

italics()

Parameters
None

Description
Use the italics method with the write or writeln methods to format and display a string in a document. In server-side JavaScript, use the write function to display the string.

Examples
The following example uses string methods to change the formatting of a string:

```
var worldString="Hello, world"

document.write(worldString.blink())
document.write("<P>" + worldString.bold())
document.write("<P>" + worldString.italics())
document.write("<P>" + worldString.strike())
```

The previous example produces the same output as the following HTML:

```
<BLINK>Hello, world</BLINK>
<P><B>Hello, world</B>
<P><I>Hello, world</I>
<P><STRIKE>Hello, world</STRIKE>
```

See also
String.blink, String.bold, String.strike

lastIndexOf

Returns the index within the calling String object of the last occurrence of the specified value, or -1 if not found. The calling string is searched backward, starting at fromIndex.

Method of	String
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
lastIndexOf(*searchValue*[, *fromIndex*])

Parameters

searchValue	A string representing the value to search for.
fromIndex	The location within the calling string to start the search from. It can be any integer between 0 and the length of the string. The default value is the length of the string.

Description
Characters in a string are indexed from left to right. The index of the first character is 0, and the index of the last character is stringName.length - 1.

```
"canal".lastIndexOf("a")   // returns 3
"canal".lastIndexOf("a",2) // returns 1
"canal".lastIndexOf("a",0) // returns -1
"canal".lastIndexOf("x")   // returns -1
```

The lastIndexOf method is case sensitive. For example, the following expression returns -1:

"Blue Whale, Killer Whale".lastIndexOf("blue")

Examples

The following example uses indexOf and lastIndexOf to locate values in the string "Brave new world."

```
var anyString="Brave new world"

// Displays 8
document.write("<P>The index of the first w from the beginning is " +
  anyString.indexOf("w"))
// Displays 10
document.write("<P>The index of the first w from the end is " +
  anyString.lastIndexOf("w"))
// Displays 6
document.write("<P>The index of 'new' from the beginning is " +
  anyString.indexOf("new"))
// Displays 6
document.write("<P>The index of 'new' from the end is " +
  anyString.lastIndexOf("new"))
```

See also

String.charAt, String.indexOf, String.split

length

The length of the string.

Property of	String
Read-only	
Implemented in	JavaScript 1.0, NES 2.0

ECMA version	ECMA-262

Description
For a null string, length is 0.

Examples
The following example displays 8 in an Alert dialog box:

```
var x="Netscape"
alert("The string length is " + x.length)
```

link

Creates an HTML hypertext link that requests another URL.

Method of	String
Implemented in	JavaScript 1.0, NES 2.0

Syntax
link(*hrefAttribute*)

Parameters

hrefAttribute	Any string that specifies the HREF attribute of the A tag; it should be a valid URL (relative or absolute).

Description
Use the link method to programmatically create a hypertext link, and then call write or writeln to display the link in a document. In server-side JavaScript, use the write function to display the link.

Links created with the link method become elements in the links array of the document object. See document.links.

Examples
The following example displays the word "Netscape" as a hypertext link that returns the user to the Netscape home page:

```
var hotText="Netscape"
var URL="http://home.netscape.com"

document.write("Click to return to " + hotText.link(URL))
```

The previous example produces the same output as the following HTML:

```
Click to return to <A HREF="http://home.netscape.com">Netscape</A>
```

match

Used to match a regular expression against a string.

Method of	String
Implemented in	JavaScript 1.2

ECMA version	ECMA-262 Edition 3

Syntax
match(*regexp*)

Parameters

regexp	Name of the regular expression. It can be a variable name or a literal.

Description
If the regular expression does not include the g flag, returns the same result that RegExp.exec would return on the regular expression and string. If the regular expression includes the g flag, returns an array of all the matches of the regular expression in the string.

Note If you execute a match simply to find true or false, use String.search or the regular expression test method.

Examples
Example 1. In the following example, match is used to find 'Chapter' followed by 1 or more numeric characters followed by a decimal point and numeric character 0 or more times. The regular expression includes the i flag so that case will be ignored.

```
<SCRIPT>
str = "For more information, see Chapter 3.4.5.1";
re = /(chapter \d+(\.\d)*)/i;
found = str.match(re);
document.write(found);
</SCRIPT>
```

This returns the array containing Chapter 3.4.5.1,Chapter 3.4.5.1,.1

'Chapter 3.4.5.1' is the first match and the first value remembered from (Chapter \d+(\.\d)*).

'.1' is the second value remembered from (\.\d).

Example 2. The following example demonstrates the use of the global and ignore case flags with match.

```
<SCRIPT>
str = "abcDdcba";
newArray = str.match(/d/gi);
document.write(newArray);
</SCRIPT>
```

The returned array contains D, d.

prototype

Represents the prototype for this class. You can use the prototype to add properties or methods to all instances of a class. For information on prototypes, see Function.prototype.

Property of	String
Implemented in	JavaScript 1.1, NES 3.0
ECMA version	ECMA-262

replace

Finds a match between a regular expression and a string, and replaces the matched substring with a new substring.

Method of	String
Implemented in	JavaScript 1.2 JavaScript 1.3: added the ability to specify a function as the second parameter.
ECMA version	ECMA-262 Edition 3

Syntax
replace(*regexp*, *newSubStr*)
replace(*regexp*, *function*)

Versions prior to JavaScript 1.3:

replace(*regexp*, *newSubStr*)

Parameters

regexp	The name of the regular expression. It can be a variable name or a literal.
newSubStr	The string to put in place of the string found with regexp.

function	A function to be invoked after the match has been performed.

Description
This method does not change the String object it is called on. It simply returns a new string.

If you want to execute a global search and replace, include the g flag in the regular expression.

Specifying a string as a parameter. The replacement string can include the following special replacement patterns:

$$	Inserts a '$'.
$&	Inserts the matched substring..
$`	Inserts the portion of the string that precedes the matched substring.
$´	Inserts the portion of the string that follows the matched substring.
$n or $nn	Where *n* or *nn* are decimal digits, inserts the *n*th parenthesized submatch string.

Specifying a function as a parameter. When you specify a function as the second parameter, the function is invoked after the match has been performed. (The use of a function in this manner is often called a lambda expression.)

In your function, you can dynamically generate the string that replaces the matched

substring. The result of the function call is used as the replacement value.

The nested function can use the matched substrings to determine the new string (newSubStr) that replaces the found substring. You get the matched substrings through the parameters of your function. The first parameter of your function holds the complete matched substring. The following *n* parameters can be used for parenthetical matches, remembered submatch strings, where *n* is the number of submatch strings in the regular expression. Finally, the last two parameters are the offset within the string where the match occurred and the string itself. For example, the following replace method returns XX.zzzz - XX , zzzz.

```
"XXzzzz".replace(/(X*)(z*)/,
                 function (str, p1, p2, offset, s) {
                   return str + " - " + p1 + " , " + p2;
                 }
               )
```

Backward Compatibility

JavaScript 1.2. You cannot specify a function to be invoked after the match has been performed.

Examples

Example 1. In the following example, the regular expression includes the global and ignore case flags which permits replace to replace each occurrence of 'apples' in the string with 'oranges.'

```
<SCRIPT>
re = /apples/gi;
str = "Apples are round, and apples are juicy.";
newstr=str.replace(re, "oranges");
document.write(newstr)
</SCRIPT>
```

This prints "oranges are round, and oranges are juicy."

Example 2. In the following example, the regular expression is defined in replace and includes the ignore case flag.

```
<SCRIPT>
str = "Twas the night before Xmas...";
newstr=str.replace(/xmas/i, "Christmas");
```

```
document.write(newstr)
</SCRIPT>
```

This prints "Twas the night before Christmas..."

Example 3. The following script switches the words in the string. For the replacement text, the script uses the $1 and $2 replacement patterns.

```
<SCRIPT LANGUAGE="JavaScript1.2">
re = /(\w+)\s(\w+)/;
str = "John Smith";
newstr = str.replace(re, "$2, $1");
document.write(newstr)
</SCRIPT>
```

This prints "Smith, John".

Example 4. The following example replaces a Fahrenheit degree with its equivalent Celsius degree. The Fahrenheit degree should be a number ending with F. The function returns the Celsius number ending with C. For example, if the input number is 212F, the function returns 100C. If the number is 0F, the function returns -17.77777777777778C.

The regular expression test checks for any number that ends with F. The number of Fahrenheit degree is accessible to your function through the parameter $1. The function sets the Celsius number based on the Fahrenheit degree passed in a string to the f2c function. f2c then returns the Celsius number. This function approximates Perl's s///e flag.

```
function f2c(x) {
  var s = String(x)
  var test = /(\d+(?:\.\d*)?)F\b/g
  return s.replace
    (test,
      function (str,p1,offset,s) {
        return ((p1-32) * 5/9) + "C";
      }
    )
}
```

search

Executes the search for a match between a regular expression and this String object.

Method of	String
Implemented in	JavaScript 1.2
ECMA version	ECMA-262 Edition 3

Syntax
search(*regexp*)

Parameters

regexp	Name of the regular expression. It can be a variable name or a literal.

Description
If successful, search returns the index of the regular expression inside the string. Otherwise, it returns -1.

When you want to know whether a pattern is found in a string use search (similar to the regular expression test method); for more information (but slower execution) use match (similar to the regular expression exec method).

Example
The following example prints a message which depends on the success of the test.

```
function testinput(re, str){
```

```
  if (str.search(re) != -1)
    midstring = " contains ";
  else
    midstring = " does not contain ";
  document.write (str + midstring + re.source);
}
```

slice

Extracts a section of a string and returns a new string.

Method of	String
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262 Edition 3

Syntax
slice(*beginslice*[, *endSlice*])

Parameters

beginSlice	The zero-based index at which to begin extraction.

endSlice	The zero-based index at which to end extraction. If omitted, slice extracts to the end of the string.

Description
slice extracts the text from one string and returns a new string. Changes to the text in one string do not affect the other string.

slice extracts up to but not including endSlice. string.slice(1,4) extracts the second character through the fourth character (characters indexed 1, 2, and 3).

As a negative index, endSlice indicates an offset from the end of the string. string.slice(2,-1) extracts the third character through the second to last character in the string.

Example
The following example uses slice to create a new string.

```
<SCRIPT>
str1="The morning is upon us. "
str2=str1.slice(3,-5)
document.write(str2)
</SCRIPT>
```

This writes:

morning is upon

small

Causes a string to be displayed in a small font, as if it were in a <SMALL> tag.

Method of	String

Implemented in	JavaScript 1.0, NES 2.0

Syntax
small()

Parameters
None

Description
Use the small method with the write or writeln methods to format and display a string in a document. In server-side JavaScript, use the write function to display the string.

Examples
The following example uses string methods to change the size of a string:

```
var worldString="Hello, world"

document.write(worldString.small())
document.write("<P>" + worldString.big())
document.write("<P>" + worldString.fontsize(7))
```

The previous example produces the same output as the following HTML:

```
<SMALL>Hello, world</SMALL>
<P><BIG>Hello, world</BIG>
<P><FONTSIZE=7>Hello, world</FONTSIZE>
```

See also
String.big, String.fontsize

split

Splits a String object into an array of strings by separating the string into substrings.

Method of	String
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262 (if separator is a string) ECMA-262 Edition 3 (if separator is a regular expression)

Syntax
split([*separator*][, *limit*])

Parameters

separator	Specifies the character to use for separating the string. The separator is treated as a string or a regular expression. If separator is omitted, the array returned contains one element consisting of the entire string.
limit	Integer specifying a limit on the number of splits to be found.

Description
The split method returns the new array.

When found, separator is removed from the string and the substrings are returned in an array. If separator is omitted, the array contains one element consisting of the entire string.

In JavaScript 1.2 or later, split has the following additions:

-
- It can take a regular expression argument, as well as a fixed string, by which to split the object string. If separator is a regular expression, any included parenthesis cause submatches to be included in the returned array.

- It can take a limit count so that the resulting array does not include trailing empty elements.

- If you specify LANGUAGE="JavaScript1.2" in the SCRIPT tag, string.split(" ") splits on any run of 1 or more white space characters including spaces, tabs, line feeds, and carriage returns. For this behavior, LANGUAGE="JavaScript1.2" must be specified in the <SCRIPT> tag.

Examples

Example 1. The following example defines a function that splits a string into an array of strings using the specified separator. After splitting the string, the function displays messages indicating the original string (before the split), the separator used, the number of elements in the array, and the individual array elements.

```
function splitString (stringToSplit,separator) {
  arrayOfStrings = stringToSplit.split(separator)
  document.write ('<P>The original string is: "' + stringToSplit + '"')
  document.write ('<BR>The separator is: "' + separator + '"')
  document.write ("<BR>The array has " + arrayOfStrings.length + " elements: ")

  for (var i=0; i < arrayOfStrings.length; i++) {
    document.write (arrayOfStrings[i] + " / ")
  }
}

var tempestString="Oh brave new world that has such people in it."
var monthString="Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec"

var space=" "
var comma=","

splitString(tempestString,space)
splitString(tempestString)
splitString(monthString,comma)
```

This example produces the following output:

The original string is: "Oh brave new world that has such people in it."
The separator is: " "
The array has 10 elements: Oh / brave / new / world / that / has / such / people / in / it. /

The original string is: "Oh brave new world that has such people in it."
The separator is: "undefined"
The array has 1 elements: Oh brave new world that has such people in it. /

The original string is: "Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec"
The separator is: ","
The array has 12 elements: Jan / Feb / Mar / Apr / May / Jun / Jul / Aug / Sep / Oct / Nov / Dec /

Example 2. Consider the following script:

```
<SCRIPT LANGUAGE="JavaScript1.2">
str="She sells seashells \nby the\n seashore"
document.write(str + "<BR>")
a=str.split(" ")
document.write(a)
</SCRIPT>
```

Using LANGUAGE="JavaScript1.2", this script produces

"She", "sells", "seashells", "by", "the", "seashore"

Without LANGUAGE="JavaScript1.2", this script splits only on single space characters, producing

"She", "sells", , , , "seashells", "by", , , "the", "seashore"

Example 3. In the following example, split looks for 0 or more spaces followed by a semicolon followed by 0 or more spaces and, when found, removes the spaces from the string. nameList is the array returned as a result of split.

```
<SCRIPT>
names = "Harry Trump ;Fred Barney; Helen Rigby ; Bill Abel ;Chris Hand ";
document.write (names + "<BR>" + "<BR>");
re = /\s*;\s*/;
nameList = names.split (re);
document.write(nameList);
</SCRIPT>
```

This prints two lines; the first line prints the original string, and the second line prints the resulting array.

Harry Trump ;Fred Barney; Helen Rigby ; Bill Abel ;Chris Hand
Harry Trump,Fred Barney,Helen Rigby,Bill Abel,Chris Hand

Example 4. In the following example, split looks for 0 or more spaces in a string and returns the first 3 splits that it finds.

```
<SCRIPT LANGUAGE="JavaScript1.2">
myVar = " Hello World. How are you doing? ";
splits = myVar.split(" ", 3);
document.write(splits)
</SCRIPT>
```

This script displays the following:

["Hello", "World.", "How"]

See also
String.charAt, String.indexOf, String.lastIndexOf

strike

Causes a string to be displayed as struck-out text, as if it were in a <STRIKE> tag.

Method of	String
Implemented in	JavaScript 1.0, NES 2.0

Syntax
strike()

Parameters
None

Description
Use the strike method with the write or writeln methods to format and display a string in a document. In server-side JavaScript, use the write function to display the string.

Examples
The following example uses string methods to change the formatting of a string:

```
var worldString="Hello, world"

document.write(worldString.blink())
document.write("<P>" + worldString.bold())
document.write("<P>" + worldString.italics())
document.write("<P>" + worldString.strike())
```

The previous example produces the same output as the following HTML:

```
<BLINK>Hello, world</BLINK>
<P><B>Hello, world</B>
<P><I>Hello, world</I>
<P><STRIKE>Hello, world</STRIKE>
```

See also
String.blink, String.bold, String.italics

sub

Causes a string to be displayed as a subscript, as if it were in a <SUB> tag.

Method of	String

Implemented in	JavaScript 1.0, NES 2.0

Syntax
sub()

Parameters
None

Description
Use the sub method with the write or writeln methods to format and display a string in a document. In server-side JavaScript, use the write function to generate the HTML.

Examples
The following example uses the sub and sup methods to format a string:

```
var superText="superscript"
var subText="subscript"

document.write("This is what a " + superText.sup() + " looks like.")
document.write("<P>This is what a " + subText.sub() + " looks like.")
```

The previous example produces the same output as the following HTML:

```
This is what a <SUP>superscript</SUP> looks like.
<P>This is what a <SUB>subscript</SUB> looks like.
```

See also
String.sup

substr

Returns the characters in a string beginning at the specified location through the specified number of characters.

Method of	String
Implemented in	JavaScript 1.0, NES 2.0

Syntax
substr(*start*[, *length*])

Parameters

start	Location at which to begin extracting characters.
length	The number of characters to extract.

Description
start is a character index. The index of the first character is 0, and the index of the last character is 1 less than the length of the string. substr begins extracting characters at start and collects length number of characters.

If start is positive and is the length of the string or longer, substr returns no characters.

If start is negative, substr uses it as a character index from the end of the string. If start is negative and abs(start) is larger than the length of the string, substr uses 0 is the start index.

If length is 0 or negative, substr returns no characters. If length is omitted, start extracts characters to the end of the string.

Example
Consider the following script:

```
<SCRIPT LANGUAGE="JavaScript1.2">

str = "abcdefghij"
document.writeln("(1,2): ", str.substr(1,2))
document.writeln("(-2,2): ", str.substr(-2,2))
document.writeln("(1): ", str.substr(1))
document.writeln("(-20, 2): ", str.substr(1,20))
document.writeln("(20, 2): ", str.substr(20,2))

</SCRIPT>
```

This script displays:

```
(1,2): bc
(-2,2): ij
(1): bcdefghij
(-20, 2): bcdefghij
(20, 2):
```

See also
substring

substring

Returns a subset of a String object.

Method of	String
Implemented in	JavaScript 1.0, NES 2.0

ECMA version	ECMA-262

Syntax
substring(*indexA, indexB*)

Parameters

indexA	An integer between 0 and 1 less than the length of the string.
indexB	An integer between 0 and 1 less than the length of the string.

Description
substring extracts characters from indexA up to but not including indexB. In particular:

-
- If indexA is less than 0, indexA is treated as if it were 0.
- If indexB is greater than stringName.length, indexB is treated as if it were stringName.length.
- If indexA equals indexB, substring returns an empty string.
- If indexB is omitted, indexA extracts characters to the end of the string.

In JavaScript 1.2, using LANGUAGE="JavaScript1.2" in the SCRIPT tag,

-
- If indexA is greater than indexB, JavaScript produces a runtime error (out of

memory).

In JavaScript 1.2, without LANGUAGE="JavaScript1.2" in the SCRIPT tag,

-
- If indexA is greater than indexB, JavaScript returns a substring beginning with indexB and ending with indexA - 1.

Examples

Example 1. The following example uses substring to display characters from the string "Netscape":

```
var anyString="Netscape"

// Displays "Net"
document.write(anyString.substring(0,3))
document.write(anyString.substring(3,0))
// Displays "cap"
document.write(anyString.substring(4,7))
document.write(anyString.substring(7,4))
// Displays "Netscap"
document.write(anyString.substring(0,7))
// Displays "Netscape"
document.write(anyString.substring(0,8))
document.write(anyString.substring(0,10))
```

Example 2. The following example replaces a substring within a string. It will replace both individual characters and substrings. The function call at the end of the example changes the string "Brave New World" into "Brave New Web".

```
function replaceString(oldS,newS,fullS) {
// Replaces oldS with newS in the string fullS
  for (var i=0; i<fullS.length; i++) {
    if (fullS.substring(i,i+oldS.length) == oldS) {
      fullS = fullS.substring(0,i)+newS+fullS.substring(i+oldS.length,fullS.length)
    }
  }
  return fullS
}

replaceString("World","Web","Brave New World")
```

Example 3. In JavaScript 1.2, using LANGUAGE="JavaScript1.2", the following script

produces a runtime error (out of memory).

```
<SCRIPT LANGUAGE="JavaScript1.2">
str="Netscape"
document.write(str.substring(0,3);
document.write(str.substring(3,0);
</SCRIPT>
```

Without LANGUAGE="JavaScript1.2", the above script prints the following:

Net Net

In the second write, the index numbers are swapped.

See also
substr

sup

Causes a string to be displayed as a superscript, as if it were in a <SUP> tag.

Method of	String
Implemented in	JavaScript 1.0, NES 2.0

Syntax
sup()

Parameters
None

Description

Use the sup method with the write or writeln methods to format and display a string in a document. In server-side JavaScript, use the write function to generate the HTML.

Examples
The following example uses the sub and sup methods to format a string:

```
var superText="superscript"
var subText="subscript"

document.write("This is what a " + superText.sup() + " looks like.")
document.write("<P>This is what a " + subText.sub() + " looks like.")
```

The previous example produces the same output as the following HTML:

```
This is what a <SUP>superscript</SUP> looks like.
<P>This is what a <SUB>subscript</SUB> looks like.
```

See also
String.sub

toLowerCase

Returns the calling string value converted to lowercase.

Method of	String
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
toLowerCase()

Parameters
None

Description
The toLowerCase method returns the value of the string converted to lowercase. toLowerCase does not affect the value of the string itself.

Examples
The following example displays the lowercase string "alphabet":

```
var upperText="ALPHABET"
document.write(upperText.toLowerCase())
```

See also
String.toUpperCase

toSource

Returns a string representing the source code of the object.

Method of	String
Implemented in	JavaScript 1.3

Syntax
toSource()

Parameters
None

Description
The toSource method returns the following values:

-
- For the built-in String object, toSource returns the following string indicating that the source code is not available:

```
function String() {
  [native code]
}
```

- For instances of String or string literals, toSource returns a string representing the source code.

This method is usually called internally by JavaScript and not explicitly in code.

toString

Returns a string representing the specified object.

Method of	String
Implemented in	JavaScript 1.1, NES 2.0
ECMA version	ECMA-262

Syntax
toString()

Parameters
None.

Description
The String object overrides the toString method of the Object object; it does not inherit Object.toString. For String objects, the toString method returns a string representation of the object.

Examples
The following example displays the string value of a String object:

```
x = new String("Hello world");
alert(x.toString())      // Displays "Hello world"
```

See also
Object.toString

toUpperCase

Returns the calling string value converted to uppercase.

Method of	String
Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
toUpperCase()

Parameters
None

Description
The toUpperCase method returns the value of the string converted to uppercase. toUpperCase does not affect the value of the string itself.

Examples
The following example displays the string "ALPHABET":

```
var lowerText="alphabet"
document.write(lowerText.toUpperCase())
```

See also
String.toLowerCase

valueOf

Returns the primitive value of a String object.

Method of	String
Implemented in	JavaScript 1.1
ECMA version	ECMA-262

Syntax
valueOf()

Parameters
None

Description

The valueOf method of String returns the primitive value of a String object as a string data type. This value is equivalent to String.toString.

This method is usually called internally by JavaScript and not explicitly in code.

Examples

```
x = new String("Hello world");
alert(x.valueOf())        // Displays "Hello world"
```

See also

String.toString, Object.valueOf

Previous **Contents** **Index** **Next**

Last Updated **September 28, 2000**

sun

A top-level object used to access any Java class in the package sun.*.

Core object	
Implemented in	JavaScript 1.1, NES 2.0

Created by
The sun object is a top-level, predefined JavaScript object. You can automatically access it without using a constructor or calling a method.

Description
The sun object is a convenience synonym for the property Packages.sun.

See also
Packages, Packages.sun

Last Updated **September 28, 2000**

Chapter 2 Chapter 2 Top-Level Properties and Functions

This chapter contains all JavaScript properties and functions not associated with any object. In the ECMA specification, these properties and functions are referred to as properties and methods of the global object.

The following table summarizes the top-level properties.

Table 2.1 Top-level properties

Property	Description
Infinity	A numeric value representing infinity.
NaN	A value representing Not-A-Number.
undefined	The value undefined.

The following table summarizes the top-level functions.

Table 2.2 Top-level functions

Function	Description
decodeURI	Decodes a URI which has been encoded with encodeURI.
decodeURIComponent	Decodes a URI which has been encoded with encodeURIComponent
encodeURI	Computes a new version of a complete URI replacing each instance of certain characters with escape sequences representing the UTF-8 encoding of the characters.
encodeURIComponent	Computes a new version of components of a URI replacing each instance of certain characters with escape sequences representing the UTF-8 encoding of the characters.
eval	Evaluates a string of JavaScript code without reference to a particular object.
isFinite	Evaluates an argument to determine whether it is a finite number.
isNaN	Evaluates an argument to determine if it is not a number.
Number	Converts an object to a number.
parseFloat	Parses a string argument and returns a floating-point number.

parseInt	Parses a string argument and returns an integer.
String	Converts an object to a string.

decodeURI

Decodes a Uniform Resource Identifier (URI) previously created by encodeURI or by a similar routine.

Core function	
Implemented in	JavaScript 1.5, NES 6.0
ECMA version	ECMA-262 Edition 3.

Syntax

decodeURI(encodedURI)

Parameters

encodedUri	A complete, encoded Uniform Resource Identifier.

Description

Replaces each escape sequence in the encoded URI with the character that it represents.

Does not decode escape sequences that could not have been introduced by encodeURI.

See also

decodeURIComponent, encodeURI, encodeURIComponent

decodeURIComponent

Decodes a Uniform Resource Identifier (URI) component previously created by encodeURIComponent or by a similar routine.

Core function	
Implemented in	JavaScript 1.5, NES 6.0
ECMA version	ECMA-262 Edition 3.

Syntax

decodeURIComponent(encodedURI)

Parameters

encodedUri	An encoded component of a Uniform Resource Identifier.

Description

Replaces each escape sequence in the encoded URI component with the character that it represents.

See also
decodeURI, encodeURI, encodeURIComponent

encodeURI

Encodes a Uniform Resource Identifier (URI) by replacing each instance of certain characters by one, two, or three escape sequences representing the UTF-8 encoding of the character.

Core function	
Implemented in	JavaScript 1.5, NES 6.0
ECMA version	ECMA-262 Edition 3.

Syntax
encodeURI(uri)

Parameters

uri	A complete Uniform Resource Identifier.

Description
Assumes that the URI is a complete URI, so does not encode reserved characters that have special meaning in the URI.

encodeURI replaces all characters except the following with the appropriate UTF-8 escape sequences:

.

reserved characters	, / ? : @ & = + $,
unescaped characters	alphabetic, decimal digits, - _ . ! ~ * ' ( )
score	#

See also
decodeURI, eval, encodeURIComponent

encodeURIComponent

Encodes a Uniform Resource Identifier (URI) component by replacing each instance of certain characters by one, two, or three escape sequences representing the UTF-8 encoding of the character.

Core function	
Implemented in	JavaScript 1.5, NES 6.0
ECMA version	ECMA-262 Edition 3.

Syntax
encodeURIComponent(uri)

Parameters

uri	A component of a Uniform Resource Identifier.

Description
Assumes that the URI is a URI component rather than a complete URI, so does not treat reserved characters as if they have special meaning and encodes them. See encodeURI for the list of reserved characters.

encodeURIComponent replaces all characters except the following with the appropriate UTF-8 escape sequences:

.

unescaped characters	alphabetic, decimal digits, - _ . ! ~ * ' ( )

score	#

See also
decodeURI, eval, encodeURI

eval

Evaluates a string of JavaScript code without reference to a particular object.

Core function	
Implemented in	JavaScript 1.0 JavaScript 1.4: eval cannot be called indirectly
ECMA version	ECMA-262

Syntax
eval(*string*)

Parameters

string	A string representing a JavaScript expression, statement, or sequence of statements. The expression can include variables and properties of existing objects.

Description
eval is a top-level function and is not associated with any object.

The argument of the eval function is a string. If the string represents an expression, eval evaluates the expression. If the argument represents one or more JavaScript statements, eval performs the statements. Do not call eval to evaluate an arithmetic expression; JavaScript evaluates arithmetic expressions automatically.

If you construct an arithmetic expression as a string, you can use eval to evaluate it at a later time. For example, suppose you have a variable x. You can postpone evaluation of an expression involving x by assigning the string value of the expression, say "3 * x + 2", to a variable, and then calling eval at a later point in your script.

If the argument of eval is not a string, eval returns the argument unchanged. In the following example, the String constructor is specified, and eval returns a String object rather than evaluating the string.

```
eval(new String("2+2")) // returns a String object containing "2+2"
eval("2+2")             // returns 4
```

You cannot indirectly use the eval function by invoking it via a name other than eval; if you do, a runtime error might occur. For example, you should not use the following code:

```
var x = 2
var y = 4
var myEval = eval
myEval("x + y")
```

Backward Compatibility

JavaScript 1.3 and earlier versions. You can use eval indirectly, although it is discouraged.

JavaScript 1.1. eval is also a method of all objects. This method is described for the

Object class.

Examples

The following examples display output using document.write. In server-side JavaScript, you can display the same output by calling the write function instead of using document.write.

Example 1. In the following code, both of the statements containing eval return 42. The first evaluates the string "x + y + 1"; the second evaluates the string "42".

```
var x = 2
var y = 39
var z = "42"
eval("x + y + 1") // returns 42
eval(z)           // returns 42
```

Example 2. In the following example, the getFieldName(n) function returns the name of the specified form element as a string. The first statement assigns the string value of the third form element to the variable field. The second statement uses eval to display the value of the form element.

```
var field = getFieldName(3)
document.write("The field named ", field, " has value of ",
  eval(field + ".value"))
```

Example 3. The following example uses eval to evaluate the string str. This string consists of JavaScript statements that open an Alert dialog box and assign z a value of 42 if x is five, and assigns 0 to z otherwise. When the second statement is executed, eval will cause these statements to be performed, and it will also evaluate the set of statements and return the value that is assigned to z.

```
var str = "if (x == 5) {alert('z is 42'); z = 42;} else z = 0; "
document.write("<P>z is ", eval(str))
```

Example 4. In the following example, the setValue function uses eval to assign the value of the variable newValue to the text field textObject:

```
function setValue (textObject, newValue) {
  eval ("document.forms[0]." + textObject + ".value") = newValue
}
```

See also

Object.eval method

Infinity

A numeric value representing infinity.

Core property	
Implemented in	JavaScript 1.3 (In previous versions, Infinity was defined only as a property of the Number object.)
ECMA version	ECMA-262

Syntax
Infinity

Description
Infinity is a top-level property and is not associated with any object.

The initial value of Infinity is Number.POSITIVE_INFINITY. The value Infinity (positive infinity) is greater than any other number including itself. This value behaves mathematically like infinity; for example, anything multiplied by Infinity is Infinity, and anything divided by Infinity is 0.

See also
Number.NEGATIVE_INFINITY , Number.POSITIVE_INFINITY

isFinite

Evaluates an argument to determine whether it is a finite number.

Core function	
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax
isFinite(*number*)

Parameters

number	The number to evaluate.

Description
isFinite is a top-level function and is not associated with any object.

You can use this method to determine whether a number is a finite number. The isFinite method examines the number in its argument. If the argument is NaN, positive infinity or negative infinity, this method returns false, otherwise it returns true.

Examples
You can check a client input to determine whether it is a finite number.

```
if(isFinite(ClientInput) == true)
{
  /* take specific steps */
}
```

See also
Number.NEGATIVE_INFINITY , Number.POSITIVE_INFINITY

isNaN

Evaluates an argument to determine if it is not a number.

Core function	
Implemented in	JavaScript 1.0: Unix only JavaScript 1.1, NES 2.0: all platforms
ECMA version	ECMA-262

Syntax
isNaN(*testValue*)

Parameters

testValue	The value you want to evaluate.

Description
isNaN is a top-level function and is not associated with any object.

The parseFloat and parseInt functions return NaN when they evaluate a value that is not a number. isNaN returns true if passed NaN, and false otherwise.

Examples
The following example evaluates floatValue to determine if it is a number and then calls a procedure accordingly:

```
floatValue=parseFloat(toFloat)

if (isNaN(floatValue)) {
  notFloat()
} else {
  isFloat()
}
```

See also
Number.NaN, parseFloat, parseInt

NaN

A value representing Not-A-Number.

Core property	

Implemented in	JavaScript 1.3 (In previous versions, NaN was defined only as a property of the Number object)
ECMA version	ECMA-262

Syntax
NaN

Description
NaN is a top-level property and is not associated with any object.

The initial value of NaN is NaN.

NaN is always unequal to any other number, including NaN itself; you cannot check for the not-a-number value by comparing to Number.NaN. Use the isNaN function instead.

Several JavaScript methods (such as the Number constructor, parseFloat, and parseInt) return NaN if the value specified in the parameter is not a number.

You might use the NaN property to indicate an error condition for a function that should return a valid number.

See also
isNaN, Number.NaN

Number

Converts the specified object to a number.

Core function	

Implemented in	JavaScript 1.2, NES 3.0
ECMA version	ECMA-262

Syntax
Number(*obj*)

Parameter

obj	An object.

Description
Number is a top-level function and is not associated with any object.

When the object is a Date object, Number returns a value in milliseconds measured from 01 January, 1970 UTC (GMT), positive after this date, negative before.

If obj is a string that does not contain a well-formed numeric literal, Number returns NaN.

Example
The following example converts the Date object to a numerical value:

```
d = new Date ("December 17, 1995 03:24:00")
alert (Number(d))
```

This displays a dialog box containing "819199440000."

See also
Number

parseFloat

Parses a string argument and returns a floating point number.

Core function	
Implemented in	JavaScript 1.0: If the first character of the string specified in parseFloat(string) cannot be converted to a number, returns NaN on Solaris and Irix and 0 on all other platforms. JavaScript 1.1, NES 2.0: Returns NaN on all platforms if the first character of the string specified in parseFloat(string) cannot be converted to a number.
ECMA version	ECMA-262

Syntax
parseFloat(*string*)

Parameters

string	A string that represents the value you want to parse.

Description
parseFloat is a top-level function and is not associated with any object.

parseFloat parses its argument, a string, and returns a floating point number. If it encounters a character other than a sign (+ or -), numeral (0-9), a decimal point, or an exponent, it returns the value up to that point and ignores that character and all succeeding characters. Leading and trailing spaces are allowed.

If the first character cannot be converted to a number, parseFloat returns NaN.

For arithmetic purposes, the NaN value is not a number in any radix. You can call the isNaN function to determine if the result of parseFloat is NaN. If NaN is passed on to arithmetic operations, the operation results will also be NaN.

Examples
The following examples all return 3.14:

```
parseFloat("3.14")
parseFloat("314e-2")
parseFloat("0.0314E+2")
var x = "3.14"
parseFloat(x)
```

The following example returns NaN:

```
parseFloat("FF2")
```

See also
isNaN, parseInt

parseInt

Parses a string argument and returns an integer of the specified radix or base.

Core function	
Implemented in	JavaScript 1.0: If the first character of the string specified in parseInt(string) cannot be converted to a number, returns NaN on Solaris and Irix and 0 on all other platforms. JavaScript 1.1, LiveWire 2.0: Returns NaN on all platforms if the first character of the string specified in parseInt(string) cannot be converted to a number.
ECMA version	ECMA-262

Syntax
parseInt(*string*[, *radix*])

Parameters

string	A string that represents the value you want to parse.
radix	An integer that represents the radix of the return value.

Description
parseInt is a top-level function and is not associated with any object.

The parseInt function parses its first argument, a string, and attempts to return an integer of the specified radix (base). For example, a radix of 10 indicates to convert to a decimal number, 8 octal, 16 hexadecimal, and so on. For radixes above 10, the letters of the alphabet indicate numerals greater than 9. For example, for hexadecimal numbers (base 16), A through F are used.

If parseInt encounters a character that is not a numeral in the specified radix, it ignores it and all succeeding characters and returns the integer value parsed up to that point. parseInt truncates numbers to integer values. Leading and trailing spaces are allowed.

If the radix is not specified or is specified as 0, JavaScript assumes the following:

-
- If the input string begins with "0x", the radix is 16 (hexadecimal).
- If the input string begins with "0", the radix is eight (octal). This feature is deprecated.
- If the input string begins with any other value, the radix is 10 (decimal).

If the first character cannot be converted to a number, parseInt returns NaN.

For arithmetic purposes, the NaN value is not a number in any radix. You can call the isNaN function to determine if the result of parseInt is NaN. If NaN is passed on to arithmetic operations, the operation results will also be NaN.

Examples
The following examples all return 15:

```
parseInt("F", 16)
parseInt("17", 8)
parseInt("15", 10)
parseInt(15.99, 10)
parseInt("FXX123", 16)
parseInt("1111", 2)
parseInt("15*3", 10)
```

The following examples all return NaN:

```
parseInt("Hello", 8)
parseInt("0x7", 10)
parseInt("FFF", 10)
```

Even though the radix is specified differently, the following examples all return 17 because the input string begins with "0x".

```
parseInt("0x11", 16)
parseInt("0x11", 0)
parseInt("0x11")
```

See also
isNaN, parseFloat, Object.valueOf

String

Converts the specified object to a string.

Core function	
Implemented in	JavaScript 1.2, NES 3.0
ECMA version	ECMA-262

Syntax
String(*obj*)

Parameter

obj	An object.

Description
String is a top-level function and is not associated with any object.

The String method converts the value of any object into a string; it returns the same value as the toString method of an individual object.

When the object is a Date object, String returns a more readable string representation of the date. Its format is: Thu Aug 18 04:37:43 Pacific Daylight Time 1983.

Example
The following example converts the Date object to a readable string.

```
D = new Date (430054663215)
alert (String(D))
```

This displays a dialog box containing "Thu Aug 18 04:37:43 GMT-0700 (Pacific Daylight Time) 1983."

See also
String

undefined

The value undefined.

Core property	
Implemented in	JavaScript 1.3
ECMA version	ECMA-262

Syntax
undefined

Description
undefined is a top-level property and is not associated with any object.

A variable that has not been assigned a value is of type undefined. A method or statement also returns undefined if the variable that is being evaluated does not have an assigned value.

You can use undefined to determine whether a variable has a value. In the following code, the variable x is not defined, and the if statement evaluates to true.

```
var x
if(x == undefined) {
  // these statements execute
}
```

undefined is also a primitive value.

Previous **Contents** **Index** **Next**

Last Updated **September 28, 2000**

Part 2 Language Elements

Chapter 3 Statements
This chapter describes all JavaScript statements. JavaScript statements consist of keywords used with the appropriate syntax. A single statement may span multiple lines. Multiple statements may occur on a single line if each statement is separated by a semicolon.

Chapter 4 Comments
This chapter describes the syntax for comments in JavaScript.

Chapter 5 Operators
JavaScript has assignment, comparison, arithmetic, bitwise, logical, string, and special operators. This chapter describes the operators and contains information about operator precedence.

Last Updated **September 28, 2000**

Chapter 3 Chapter 3 Statements

This chapter describes all JavaScript statements. JavaScript statements consist of keywords used with the appropriate syntax. A single statement may span multiple lines. Multiple statements may occur on a single line if each statement is separated by a semicolon.

Syntax conventions: All keywords in syntax statements are in bold. Words in italics represent user-defined names or statements. Any portions enclosed in square brackets, [ ], are optional. {statements} indicates a block of statements, which can consist of zero or more statements delimited by a curly braces { }.

The following table lists statements available in JavaScript.

Table 3.1 JavaScript statements.

break	Terminates the current while or for loop and transfers program control to the statement following the terminated loop.
const	Declares a global constant, optionally initializing it to a value.
continue	Terminates execution of the block of statements in a while or for loop, and continues execution of the loop with the next iteration.
do...while	Executes the specified statements until the test condition evaluates to false. Statements execute at least once.

export	Allows a signed script to provide properties, functions, and objects to other signed or unsigned scripts.
for	Creates a loop that consists of three optional expressions, enclosed in parentheses and separated by semicolons, followed by a block of statements executed in the loop.
for...in	Iterates a specified variable over all the properties of an object. For each distinct property, JavaScript executes the specified statements.
function	Declares a function with the specified parameters. Acceptable parameters include strings, numbers, and objects.
if...else	Executes a set of statements if a specified condition is true. If the condition is false, another set of statements can be executed.
import	Allows a script to import properties, functions, and objects from a signed script that has exported the information.
label	Provides an identifier that can be used with break or continue to indicate where the program should continue execution.
return	Specifies the value to be returned by a function.
switch	Allows a program to evaluate an expression and attempt to match the expression's value to a case label.

throw	Throws a user-defined exception.
try...catch	Marks a block of statements to try, and specifies a response should an exception be thrown.
var	Declares a variable, optionally initializing it to a value.
while	Creates a loop that evaluates an expression, and if it is true, executes a block of statements. The loop then repeats, as long as the specified condition is true.
with	Establishes the default object for a set of statements.

break

Use the break statement to terminate a loop, switch, or label statement.

Terminates the current loop, switch, or label statement and transfers program control to the statement following the terminated loop.

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262 (for the unlabeled version) ECMA-262, Edition 3 (for the labeled version)

Syntax

break [*label*]

Parameter

label	Identifier associated with the label of the statement.

Description

The break statement includes an optional label that allows the program to break out of a labeled statement. The statements in a labeled statement can be of any type.

Examples

The following function has a break statement that terminates the while loop when e is 3, and then returns the value 3 * x.

```
function testBreak(x) {
  var i = 0;
  while (i < 6) {
    if (i == 3)
      break;
    i++;
  }
  return i*x;
}
```

See also

continue, label, switch

const

Declares a readonly, named constant.

Implemented in	JavaScript 1.5, NES 6.0 (Netscape extension, C engine only),

Syntax
const *constname* [= *value*] [..., *constname* [= *value*]]

Parameters

varname	Constant name. It can be any legal identifier.
value	Value of the constant and can be any legal expression.

Description
Creates a constant that can be global or local to the function in which it is declared. Constants follow the same scope rules as variables.

The value of a constant cannot change through re-assignment, and a constant cannot be re-declared.

A constant cannot share the same name as a function or variable in the same scope.

Examples
The script:

```
const a = 7;

document.writeln("a is " + a + ".");
```

produces the output:

a is 7.

continue

Restarts a while, do-while, for, or label statement.

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262 (for the unlabeled version) ECMA-262, Edition 3 (for the labeled version)

Syntax
continue [*label*]

Parameter

label	Identifier associated with the label of the statement.

Description
In contrast to the break statement, continue does not terminate the execution of the loop entirely: instead,

- In a while loop, it jumps back to the condition.
- In a for loop, it jumps to the update expression.

The continue statement can now include an optional label that allows the program to terminate execution of a labeled statement and continue to the specified labeled statement. This type of continue must be in a looping statement identified by the label used by continue.

Examples

Example 1. The following example shows a while loop that has a continue statement that executes when the value of i is 3. Thus, n takes on the values 1, 3, 7, and 12.

```
i = 0;
n = 0;
while (i < 5) {
  i++;
  if (i == 3)
    continue;
  n += i;
}
```

Example 2. In the following example, a statement labeled checkiandj contains a statement labeled checkj. If continue is encountered, the program continues at the top of the checkj statement. Each time continue is encountered, checkj reiterates until its condition returns false. When false is returned, the remainder of the checkiandj statement is completed. checkiandj reiterates until its condition returns false. When false is returned, the program continues at the statement following checkiandj.

If continue had a label of checkiandj, the program would continue at the top of the checkiandj statement.

```
checkiandj :
while (i<4) {
  document.write(i + "<BR>");
  i+=1;

  checkj :
  while (j>4) {
    document.write(j + "<BR>");
    j-=1;
    if ((j%2)==0)
```

```
        continue checkj;
      document.write(j + " is odd.<BR>");
    }
    document.write("i = " + i + "<br>");
    document.write("j = " + j + "<br>");
}
```

See also
break, label

do...while

Executes the specified statements until the test condition evaluates to false. Statements execute at least once.

Implemented in	JavaScript 1.2, NES 3.0
ECMA Version	ECMA 262, Edition 3

Syntax

```
do
   statements
while (condition);
```

Parameters

statements	Block of statements that is executed at least once and is re-executed each time the condition evaluates to true.
condition	Evaluated after each pass through the loop. If condition evaluates to true, the statements in the preceding block are re-executed. When condition evaluates to false, control passes to the statement following do while.

Examples
In the following example, the do loop iterates at least once and reiterates until i is no longer less than 5.

```
do {
  i+=1;
  document.write(i);
} while (i<5);
```

export

Allows a signed script to provide properties, functions, and objects to other signed or unsigned scripts.

This feature is not in ECMA 262, Edition 3.

Implemented in	JavaScript 1.2, NES 3.0

Syntax
export name1, name2, ..., nameN
export *

Parameters

nameN	List of properties, functions, and objects to be exported.
*	Exports all properties, functions, and objects from the script.

Description
Typically, information in a signed script is available only to scripts signed by the same principals. By exporting properties, functions, or objects, a signed script makes this information available to any script (signed or unsigned). The receiving script uses the companion import statement to access the information.

See also
import

for

Creates a loop that consists of three optional expressions, enclosed in parentheses and separated by semicolons, followed by a block of statements executed in the loop.

Implemented in	JavaScript 1.0, NES 2.0

ECMA version	ECMA-262

Syntax

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

Parameters

initial-expression	Statement or variable declaration. Typically used to initialize a counter variable. This expression may optionally declare new variables with the var keyword. These variables are local to the function, not to the loop.
condition	Evaluated on each pass through the loop. If this condition evaluates to true, the statements in statements are performed. This conditional test is optional. If omitted, the condition always evaluates to true.
increment-expression	Generally used to update or increment the counter variable.
statements	Block of statements that are executed as long as condition evaluates to true. This can be a single statement or multiple statements. Although not required, it is good practice to indent these statements from the beginning of the for statement.

Examples

The following for statement starts by declaring the variable i and initializing it to 0. It checks that i is less than nine, performs the two succeeding statements, and increments i by 1 after each pass through the loop.

```
for (var i = 0; i < 9; i++) {
  n += i;
  myfunc(n);
}
```

for...in

Iterates a specified variable over all the properties of an object. For each distinct property, JavaScript executes the specified statements.

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax

```
for (variable in object) {
  statements
}
```

Parameters

variable	Variable to iterate over every property, optionally declared with the var keyword. This variable is local to the function, not to the loop.
object	Object for which the properties are iterated.
statements	Specifies the statements to execute for each property.

Examples
The following function takes as its argument an object and the object's name. It then iterates over all the object's properties and returns a string that lists the property names and their values.

```
function show_props(obj, objName) {
  var result = "";
  for (var i in obj) {
    result += objName + "." + i + " = " + obj[i] + "\n";
  }
  return result;
}
```

function

Declares a function with the specified parameters. Acceptable parameters include strings, numbers, and objects.

Implemented in	JavaScript 1.0, NES 2.0 JavaScript 1.5, NES 6.0: added conditional function declarations (Netscape extension).
ECMA version	ECMA-262

Syntax

function *name*([*param*] [, *param*] [..., *param*]) {
  statements
}

You can also define functions using the Function constructor and the function operator; see Function and function.

Parameters

name	The function name.
param	The name of an argument to be passed to the function. A function can have up to 255 arguments.
statements	The statements which comprise the body of the function.

Description

To return a value, the function must have a return statement that specifies the value to

return.

A function created with the function statement is a Function object and has all the properties, methods, and behavior of Function objects. See Function for detailed information on functions.

Netscape supports conditional function declarations, whereby a function can be declared based on the evaluation of a condition. If the condition evaluates to true, the function is declared. Otherwise it is not declared.

A function can also be declared inside an expression. In this case the function is usually anonymous. See page 254.

Examples

The following code declares a function that returns the total dollar amount of sales, when given the number of units sold of products a, b, and c.

```
function calc_sales(units_a, units_b, units_c) {
  return units_a*79 + units_b*129 + units_c*699
}
```

In the following script, the one function is always declared. The zero function is declared because 'if(1)' evaluates to true:

```
<SCRIPT language="JavaScript1.5">
<!--
function one()
  document.writeln("This is one.");
  if (1)
    function zero()
    {
      document.writeln("This is zero.");
    }
}
</SCRIPT>
```

However, if the script is changed so that the condition becomes 'if (0)', function zero is not declared and cannot be invoked on the page.

See also

Function, function

if...else

Executes a set of statements if a specified condition is true. If the condition is false, another set of statements can be executed.

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax

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

Parameters

condition	Can be any JavaScript expression that evaluates to true or false. Parentheses are required around the condition. If condition evaluates to true, the statements in statements1 are executed.

statements1, statements2	Can be any JavaScript statements, including further nested if statements. Multiple statements must be enclosed in braces.

Description
You should not use simple assignments in a conditional statement. For example, do not use the following code:

```
if(x = y)
{
  /* do the right thing */
}
```

If you need to use an assignment in a conditional statement, put additional parentheses around the assignment. For example, use if((x = y)).

Examples

```
if (cipher_char == from_char) {
  result = result + to_char
  x++}
else
  result = result + clear_char
```

import

Allows a script to import properties, functions, and objects from a signed script that has exported the information.

This feature is not in ECMA 262, Edition 3.

Implemented in	JavaScript 1.2, NES 3.0

Syntax
import *objectName.name1, objectName.name2, ..., objectName.nameN*
import *objectName.**

Parameters

objectName	Name of the object that will receive the imported names.
name1, name2, nameN	List of properties, functions, and objects to import from the export file.
*	Imports all properties, functions, and objects from the export script.

Description
The objectName parameter is the name of the object that will receive the imported names. For example, if f and p have been exported, and if obj is an object from the importing script, the following code makes f and p accessible in the importing script as properties of obj.

import obj.f, obj.p

Typically, information in a signed script is available only to scripts signed by the same principals. By exporting (using the export statement) properties, functions, or objects, a signed script makes this information available to any script (signed or unsigned). The receiving script uses the import statement to access the information.

The script must load the export script into a window, frame, or layer before it can import and use any exported properties, functions, and objects.

See also
export

label

Provides a statement with an identifier that lets you refer to it using a break or continue statement.

Implemented in	JavaScript 1.2, NES 3.0
ECMA version	ECMA 262, Edition 3

For example, you can use a label to identify a loop, and then use the break or continue statements to indicate whether a program should interrupt the loop or continue its execution.

Syntax
label :
  statement

Parameter

label	Any JavaScript identifier that is not a reserved word.
statement	Statements. break can be used with any labeled statement, and continue can be used with looping labeled statements.

Examples
For an example of a label statement using break, see break. For an example of a label statement using continue, see continue.

See also
break, continue

return

Specifies the value to be returned by a function.

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
return *expression;*

Parameters

expression	The expression to return.

Examples
The following function returns the square of its argument, x, where x is a number.

```
function square(x) {
  return x * x;
}
```

switch

Allows a program to evaluate an expression and attempt to match the expression's value to a case label.

Implemented in	JavaScript 1.2, NES 3.0
ECMA version	ECMA-262, Edition 3

Syntax

```
switch (expression){
  case label :
    statements;
    break;
  case label :
    statements;
    break;
```

```
  ...
  default : statements;
}
```

Parameters

expression	Value matched against label.
label	Identifier used to match against expression.
statements	Block of statements that is executed once if expression matches label.

Description

If a match is found, the program executes the associated statement. If multiple cases match the provided value, the first case that matches is selected, even if the cases are not equal to each other.

The program first looks for a label matching the value of expression and then executes the associated statement. If no matching label is found, the program looks for the optional default statement, and if found, executes the associated statement. If no default statement is found, the program continues execution at the statement following the end of switch.

The optional break statement associated with each case label ensures that the program breaks out of switch once the matched statement is executed and continues execution at the statement following switch. If break is omitted, the program continues execution at the next statement in the switch statement.

Examples

In the following example, if expression evaluates to "Bananas", the program matches the value with case "Bananas" and executes the associated statement. When break is encountered, the program breaks out of switch and executes the statement following switch. If break were omitted, the statement for case "Cherries" would also be executed.

```
switch (i) {
  case "Oranges" :
    document.write("Oranges are $0.59 a pound.<BR>");
    break;
  case "Apples" :
    document.write("Apples are $0.32 a pound.<BR>");
    break;
  case "Bananas" :
    document.write("Bananas are $0.48 a pound.<BR>");
    break;
  case "Cherries" :
    document.write("Cherries are $3.00 a pound.<BR>");
    break;
  default :
    document.write("Sorry, we are out of " + i + ".<BR>");
}
document.write("Is there anything else you'd like?<BR>");
```

throw

Throws a user-defined exception.

Implemented in	JavaScript 1.4
ECMA version	ECMA-262, Edition 3

Syntax

throw *expression;*

Parameters

expression	The value to throw.

Description

Use the throw statement to throw an exception. When you throw an exception, an expression specifies the value of the exception. The following code throws several exceptions.

```
throw "Error2";    // generates an exception with a string value
throw 42;          // generates an exception with the value 42
throw true;        // generates an exception with the value true
```

Examples

Example 1: Throw an object. You can specify an object when you throw an exception. You can then reference the object's properties in the catch block. The following example creates an object myUserException of type UserException and uses it in a throw statement.

```
function UserException (message) {
  this.message=message;
  this.name="UserException";
}
function getMonthName (mo) {
  mo=mo-1; // Adjust month number for array index (1=Jan, 12=Dec)
  var months=new Array("Jan","Feb","Mar","Apr","May","Jun","Jul",
    "Aug","Sep","Oct","Nov","Dec");
  if (months[mo] != null) {
    return months[mo];
  } else {
    myUserException=new UserException("InvalidMonthNo");
```

```
    throw myUserException;
  }
}

try {
  // statements to try;
  monthName=getMonthName(myMonth)
}
catch (e) {
  monthName="unknown";
  logMyErrors(e.message,e.name); // pass exception object to err handler
}
```

Example 2: Throw an object. The following example tests an input string for a U.S. zip code. If the zip code uses an invalid format, the throw statement throws an exception by creating an object of type ZipCodeFormatException.

```
/*
 * Creates a ZipCode object.
 *
 * Accepted formats for a zip code are:
 *    12345
 *    12345-6789
 *    123456789
 *    12345 6789
 *
 * If the argument passed to the ZipCode constructor does not
 * conform to one of these patterns, an exception is thrown.
 */

function ZipCode(zip) {
  zip = new String(zip);
  pattern = /[0-9]{5}([- ]?[0-9]{4})?/;
  if (pattern.test(zip)) {
    // zip code value will be the first match in the string
    this.value = zip.match(pattern)[0];
    this.valueOf = function (){return this.value};
    this.toString = function (){return String(this.value)};
  } else {
    throw new ZipCodeFormatException(zip);
  }
}

function ZipCodeFormatException(value) {
```

```
  this.value = value;
  this.message =
    "does not conform to the expected format for a zip code";
  this.toString =
    function (){return this.value + this.message};
}

/*
 * This could be in a script that validates address data
 * for US addresses.
 */

var ZIPCODE_INVALID = -1;
var ZIPCODE_UNKNOWN_ERROR = -2;

function verifyZipCode(z) {
  try {
    z = new ZipCode(z);
  }
  catch (e) {
    if (e instanceof ZipCodeFormatException) {
      return ZIPCODE_INVALID;
    }
    else {
      return ZIPCODE_UNKNOWN_ERROR;
    }
  }
  return z;
}

a=verifyZipCode(95060);        // returns 95060
b=verifyZipCode(9560;)         // returns -1
c=verifyZipCode("a");          // returns -1
d=verifyZipCode("95060");      // returns 95060
e=verifyZipCode("95060 1234");  // returns 95060 1234
```

Example 3: Rethrow an exception. You can use throw to rethrow an exception after you catch it. The following example catches an exception with a numeric value and rethrows it if the value is over 50. The rethrown exception propagates up to the enclosing function or to the top level so that the user sees it.

```
try {
  throw n // throws an exception with a numeric value
}
```

```
catch (e) {
  if (e <= 50) {
    // statements to handle exceptions 1-50
  }
  else {
    // cannot handle this exception, so rethrow
    throw e
  }
}
```

See also
try...catch

try...catch

Marks a block of statements to try, and specifies a response should an exception be thrown.

Implemented in	JavaScript 1.4 JavaScript 1.5, NES 6.0: added multiple catch clauses (Netscape extension).
ECMA version	ECMA-262, Edition 3

Syntax

```
try {
  statements
}
[catch (exception_var if expression)
  {statements}] . . .
[catch (exception_var) {statements}]
[finally {statements}]
```

Parameters

statements	Block of statements that executes once. The statements can be declarative statements (such as var) or executable statements (such as for).
catch	A block of statements to be executed if an exception is thrown in the try block.
exception_var	An identifier to hold an exception object.
expression	A test expression.
finally	A block of statements that is executed before the try...catch statement completes. This block of statements executes whether or not an exception was thrown or caught.

Description

The try...catch statement consists of a try block, which contains one or more statements, and one or more catch blocks, containing statements that specify what to do if an exception is thrown in the try block. That is, you want the try block to succeed, and if it does not succeed, you want control to pass to the catch block. If any statement within the try block (or in a function called from within the try block) throws an exception, control immediately shifts to the catch block. If no exception is thrown in the try block succeed, the catch block is skipped. The finally block executes after the try and catch blocks execute but before the statements following the try...catch statement.

You can nest one or more try...catch statements. If an inner try...catch statement does not have a catch block, the enclosing try...catch statement's catch block is entered.

You also use the try...catch statement to handle Java exceptions. See the *Core JavaScript Guide* for information on Java exceptions.

Unconditional catch Block. When a single, unconditional catch block is used, the catch block entered when any exception is thrown. For example, the following code throws an exception. When the exception occurs, control transfers to the catch block.

```
try {
  throw "myException" // generates an exception
}
catch (e) {
  // statements to handle any exceptions
  logMyErrors(e) // pass exception object to error handler
}
```

Conditional catch Blocks. You can also use one or more conditional catch blocks to handle specific exceptions. In this case, the appropriate catch block is entered when the specified exception is thrown. In the following example, code in the try block can potentially throw three exceptions: TypeError, RangeError, and EvalError. When an exception occurs, control transfers to the appropriate catch block. If the exception is not one of the specified exceptions, control transfers to the unconditional catch block at the end. If you use an unconditional catch block with one or more conditional catch blocks, the unconditional catch block must be specified last.

```
try {
  myroutine(); // may throw three exceptions
}
catch (e if e instanceof TypeError) {
  // statements to handle TypeError exceptions
}

catch (e if e instanceof RangeError) {
  // statements to handle RangeError exceptions
}

catch (e if e instanceof EvalError) {
  // statements to handle EvalError exceptions
}

catch (e){
```

```
  // statements to handle any unspecified exceptions
  logMyErrors(e) // pass exception object to error handler
}
```

The exception Identifier. When an exception is thrown in the try block, the exception_var holds the value specified by the throw statement; you can use this identifier to get information about the exception that was thrown. JavaScript creates this identifier when the catch block is entered; the identifier lasts only for the duration of the catch block; after the catch block finishes executing, the identifier is no longer available.

The finally Block. The finally block contains statements to execute after the try and catch blocks execute but before the statements following the try...catch statement. The finally block executes whether or not an exception is thrown. If an exception is thrown, the statements in the finally block execute even if no catch block handles the exception.

You can use the finally block to make your script fail gracefully when an exception occurs; for example, you may need to release a resource that your script has tied up. The following example opens a file and then executes statements that use the file (server-side JavaScript allows you to access files). If an exception is thrown while the file is open, the finally block closes the file before the script fails.

```
openMyFile()
try {
  // tie up a resource
  writeMyFile(theData)
}
finally {
  closeMyFile() // always close the resource
}
```

Examples
See the examples for throw.

See also
throw

var

Declares a variable, optionally initializing it to a value.

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
var *varname* [= *value*] [..., *varname* [= *value*]]

Parameters

varname	Variable name. It can be any legal identifier.
value	Initial value of the variable and can be any legal expression.

Description
The scope of a variable is the current function or, for variables declared outside a function, the current application.

Using var outside a function is optional but recommended; you can declare a variable by simply assigning it a value. However, it is good style to use var, and it is necessary in functions in the following situations:

-

- If a global variable of the same name exists.
- If recursive or multiple functions use variables with the same name.

Examples
var num_hits = 0, cust_no = 0

while

Creates a loop that evaluates an expression, and if it is true, executes a block of statements. The loop then repeats, as long as the specified condition is true.

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
while (*condition*) {
  statements
}

Parameters

condition	Evaluated before each pass through the loop. If this condition evaluates to true, the statements in the succeeding block are performed. When condition evaluates to false, execution continues with the statement following statements.
statements	Block of statements that are executed as long as the condition evaluates to true. Although not required, it is good practice to indent these statements from the beginning of the statement.

Examples
The following while loop iterates as long as n is less than three.

```
n = 0;
x = 0;
while(n < 3) {
  n ++;
  x += n;
}
```

Each iteration, the loop increments n and adds it to x. Therefore, x and n take on the following values:

-
- After the first pass: n = 1 and x = 1
- After the second pass: n = 2 and x = 3
- After the third pass: n = 3 and x = 6

After completing the third pass, the condition n < 3 is no longer true, so the loop terminates.

with

Establishes the default object for a set of statements.

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax

with (*object*){
  statements
}

Parameters

object	Specifies the default object to use for the statements. The parentheses around object are required.
statements	Any block of statements.

Description

JavaScript looks up any unqualified names within the set of statements to determine if the names are properties of the default object. If an unqualified name matches a property, then the property is used in the statement; otherwise, a local or global variable is used.

Note that using a with statement will significantly slow down your code. Do not use it when performance is critical.

Examples

The following with statement specifies that the Math object is the default object. The statements following the with statement refer to the PI property and the cos and sin methods, without specifying an object. JavaScript assumes the Math object for these references.

```
var a, x, y
var r=10
with (Math) {
  a = PI * r * r
  x = r * cos(PI)
  y = r * sin(PI/2)
}
```

Previous **Contents** **Index** **Next**

Last Updated **September 28, 2000**

Chapter 4 Chapter 4 Comments

This chapter describes the syntax for comments, which can appear anywhere between tokens.

comment

Notations by the author to explain what a script does. Comments are ignored by the interpreter.

Implemented in	JavaScript 1.0, NES 2.0
ECMA version	ECMA-262

Syntax
// comment text
/* multiple line comment text */

Description
JavaScript supports Java-style comments:

-
- Comments on a single line are preceded by a double-slash (//).
- Comments that span multiple lines are preceded by a /* and followed by a */.

Examples
// This is a single-line comment.

```
/* This is a multiple-line comment. It can be of any length, and
you can put whatever you want here. */
```

Previous **Contents** **Index** **Next**

Last Updated **September 28, 2000**

Chapter 5 Chapter 5 Operators

JavaScript has assignment, comparison, arithmetic, bitwise, logical, string, and special operators. This chapter describes the operators and contains information about operator precedence.

The following table summarizes the JavaScript operators.

Table 5.1 JavaScript operators.

Operator category	Operator	Description
Arithmetic Operators	+	(Addition) Adds 2 numbers.
	++	(Increment) Adds one to a variable representing a number (returning either the new or old value of the variable).
	-	(Unary negation, subtraction) As a unary operator, negates the value of its argument. As a binary operator, subtracts 2 numbers.
	--	(Decrement) Subtracts one from a variable representing a number (returning either the new or old value of the variable).
	*	(Multiplication) Multiplies 2 numbers.

	/	(Division) Divides 2 numbers.
	%	(Modulus) Computes the integer remainder of dividing 2 numbers.
String Operators	+	(String addition) Concatenates 2 strings.
	+=	Concatenates 2 strings and assigns the result to the first operand.
Logical Operators	&&	(Logical AND) Returns the first operand if it can be converted to false; otherwise, returns the second operand. Thus, when used with Boolean values, && returns true if both operands are true; otherwise, returns false.
	||	(Logical OR) Returns the first operand if it can be converted to true; otherwise, returns the second operand. Thus, when used with Boolean values, || returns true if either operand is true; if both are false, returns false.
	!	(Logical NOT) Returns false if its single operand can be converted to true; otherwise, returns true.
Bitwise Operators	&	(Bitwise AND) Returns a one in each bit position if bits of both operands are ones.
	^	(Bitwise XOR) Returns a one in a bit position if bits of one but not both operands are one.
	|	(Bitwise OR) Returns a one in a bit if bits of either operand is one.

	~	(Bitwise NOT) Flips the bits of its operand.
	<<	(Left shift) Shifts its first operand in binary representation the number of bits to the left specified in the second operand, shifting in zeros from the right.
	>>	(Sign-propagating right shift) Shifts the first operand in binary representation the number of bits to the right specified in the second operand, discarding bits shifted off.
	>>>	(Zero-fill right shift) Shifts the first operand in binary representation the number of bits to the right specified in the second operand, discarding bits shifted off, and shifting in zeros from the left.
Assignment Operators	=	Assigns the value of the second operand to the first operand.
	+=	Adds 2 numbers and assigns the result to the first.
	-=	Subtracts 2 numbers and assigns the result to the first.
	*=	Multiplies 2 numbers and assigns the result to the first.
	/=	Divides 2 numbers and assigns the result to the first.

	%=	Computes the modulus of 2 numbers and assigns the result to the first.
	&=	Performs a bitwise AND and assigns the result to the first operand.
	^=	Performs a bitwise XOR and assigns the result to the first operand.
	|=	Performs a bitwise OR and assigns the result to the first operand.
	<<=	Performs a left shift and assigns the result to the first operand.
	>>=	Performs a sign-propagating right shift and assigns the result to the first operand.
	>>>=	Performs a zero-fill right shift and assigns the result to the first operand.
Comparison Operators	==	Returns true if the operands are equal.
	!=	Returns true if the operands are not equal.
	===	Returns true if the operands are equal and of the same type.
	!==	Returns true if the operands are not equal and/or not of the same type.

	>	Returns true if the left operand is greater than the right operand.
	>=	Returns true if the left operand is greater than or equal to the right operand.
	<	Returns true if the left operand is less than the right operand.
	<=	Returns true if the left operand is less than or equal to the right operand.
Special Operators	?:	Performs a simple "if...then...else".
	,	Evaluates two expressions and returns the result of the second expression.
	delete	Deletes an object, an object's property, or an element at a specified index in an array.
	function	Defines an anonymous function.
	in	Returns true if the specified property is in the specified object.
	instanceof	Returns true if the specified object is of the specified object type.

new	Creates an instance of a user-defined object type or of one of the built-in object types.
this	Keyword that you can use to refer to the current object.
typeof	Returns a string indicating the type of the unevaluated operand.
void	Specifies an expression to be evaluated without returning a value.

Assignment Operators

An assignment operator assigns a value to its left operand based on the value of its right operand.

Implemented in	JavaScript 1.0
ECMA version	ECMA-262

The basic assignment operator is equal (=), which assigns the value of its right operand to its left operand. That is, x = y assigns the value of y to x. The other assignment operators are usually shorthand for standard operations, as shown in the following table.

Table 5.2 Assignment operators

Shorthand operator	Meaning
x += y	x = x + y
x -= y	x = x - y
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y
x <<= y	x = x << y
x >>= y	x = x >> y
x >>>= y	x = x >>> y
x &= y	x = x & y

x ^= y	x = x ^ y
x |= y	x = x | y

In unusual situations, the assignment operator is not identical to the Meaning expression in Table 5.2. When the left operand of an assignment operator itself contains an assignment operator, the left operand is evaluated only once. For example:

```
a[i++] += 5 //i is evaluated only once
a[i++] = a[i++] + 5 //i is evaluated twice
```

Comparison Operators

A comparison operator compares its operands and returns a logical value based on whether the comparison is true.

Implemented in	JavaScript 1.0 JavaScript 1.3: Added the === and !== operators. JavaScript 1.4: Deprecated == for comparison of two JSObject objects. Use the JSObject.equals method.
ECMA version	ECMA-262 includes all comparison operators except === and !==. ECMA-262 Edition 3 adds === and !==.

The operands can be numerical or string values. Strings are compared based on standard lexicographical ordering, using Unicode values.

A Boolean value is returned as the result of the comparison.

-
- Two strings are equal when they have the same sequence of characters, same length, and same characters in corresponding positions.
- Two numbers are equal when they are numerically equal (have the same number value). NaN is not equal to anything, including NaN. Positive and negative zeros are equal.
- Two objects are equal if they refer to the same Object.
- Two Boolean operands are equal if they are both true or false.
- Null and Undefined types are == (but not ===).

The following table describes the comparison operators.

Table 5.3 Comparison operators

Operator	Description	Examples returning true[1]
Equal (==)	Returns true if the operands are equal. If the two operands are not of the same type, JavaScript attempts to convert the operands to an appropriate type for the comparison.	3 == var1 "3" == var1 3 == '3'
Not equal (!=)	Returns true if the operands are not equal. If the two operands are not of the same type, JavaScript attempts to convert the operands to an appropriate type for the comparison.	var1 != 4 var1 != "3"
Strict equal (===)	Returns true if the operands are equal and of the same type.	3 === var1

Strict not equal (!==)	Returns true if the operands are not equal and/or not of the same type.	var1 !== "3" 3 !== '3'
Greater than (>)	Returns true if the left operand is greater than the right operand.	var2 > var1
Greater than or equal (>=)	Returns true if the left operand is greater than or equal to the right operand.	var2 >= var1 var1 >= 3
Less than (<)	Returns true if the left operand is less than the right operand.	var1 < var2
Less than or equal (<=)	Returns true if the left operand is less than or equal to the right operand.	var1 <= var2 var2 <= 5

[1] These examples assume that var1 has been assigned the value 3 and var2 has been assigned the value 4.

Using the Equality Operators

The standard equality operators (== and !=) compare two operands without regard to their type. The strict equality operators (=== and !==) perform equality comparisons on operands of the same type. Use strict equality operators if the operands must be of a specific type as well as value or if the exact type of the operands is important. Otherwise, use the standard equality operators, which allow you to compare the identity of two operands even if they are not of the same type.

When type conversion is needed, JavaScript converts String, Number, Boolean, or Object operands as follows.

-
- When comparing a number and a string, the string is converted to a number value. JavaScript attempts to convert the string numeric literal to a Number type

value. First, a mathematical value is derived from the string numeric literal. Next, this value is rounded to nearest Number type value.

- If one of the operands is Boolean, the Boolean operand is converted to 1 if it is true and +0 if it is false.

- If an object is compared with a number or string, JavaScript attempts to return the default value for the object. Operators attempt to convert the object to a primitive value, a String or Number value, using the valueOf and toString methods of the objects. If this attempt to convert the object fails, a runtime error is generated.

You cannot use the standard equality operator (==) to compare instances of JSObject. Use the JSObject.equals method for such comparisons.

Backward Compatibility
The behavior of the standard equality operators (== and !=) depends on the JavaScript version.

JavaScript 1.3 and earlier versions. You can use either the standard equality operator (==) or JSObject.equals to compare instances of JSObject.

JavaScript 1.2. The standard equality operators (== and !=) do not perform a type conversion before the comparison is made. The strict equality operators (=== and !==) are unavailable.

JavaScript 1.1 and earlier versions. The standard equality operators (== and !=) perform a type conversion before the comparison is made. The strict equality operators (=== and !==) are unavailable.

Arithmetic Operators

Arithmetic operators take numerical values (either literals or variables) as their operands and return a single numerical value. The standard arithmetic operators are addition (+), subtraction (-), multiplication (*), and division (/).

Implemented in	JavaScript 1.0

ECMA version	ECMA-262

These operators work as they do in most other programming languages, except the / operator returns a floating-point division in JavaScript, not a truncated division as it does in languages such as C or Java. For example:

```
1/2 //returns 0.5 in JavaScript
1/2 //returns 0 in Java
```

% (Modulus)

The modulus operator is used as follows:

var1 % *var2*

The modulus operator returns the first operand modulo the second operand, that is, var1 modulo var2, in the preceding statement, where var1 and var2 are variables. The modulo function is the integer remainder of dividing var1 by var2. For example, 12 % 5 returns 2.

++ (Increment)

The increment operator is used as follows:

var ++ or ++*var*

This operator increments (adds one to) its operand and returns a value. If used postfix, with operator after operand (for example, x++), then it returns the value before incrementing. If used prefix with operator before operand (for example, ++x), then it returns the value after incrementing.

For example, if x is three, then the statement y = x++ sets y to 3 and increments x to 4. If x is 3, then the statement y = ++x increments x to 4 and sets y to 4.

-- (Decrement)

The decrement operator is used as follows:

var -- or --*var*

This operator decrements (subtracts one from) its operand and returns a value. If used postfix (for example, x--), then it returns the value before decrementing. If used prefix (for example, --x), then it returns the value after decrementing.

For example, if x is three, then the statement y = x-- sets y to 3 and decrements x to 2. If x is 3, then the statement y = --x decrements x to 2 and sets y to 2.

- (Unary Negation)

The unary negation operator precedes its operand and negates it. For example, y = -x negates the value of x and assigns that to y; that is, if x were 3, y would get the value -3 and x would retain the value 3.

Bitwise Operators

Bitwise operators treat their operands as a set of 32 bits (zeros and ones), rather than as decimal, hexadecimal, or octal numbers. For example, the decimal number nine has a binary representation of 1001. Bitwise operators perform their operations on such binary representations, but they return standard JavaScript numerical values.

The following table summarizes JavaScript's bitwise operators:

Table 5.4 Bitwise operators

Operator	Usage	Description
Bitwise AND	a & b	Returns a one in each bit position for which the corresponding bits of both operands are ones.
Bitwise OR	a | b	Returns a one in each bit position for which the corresponding bits of either or both operands are ones.

Bitwise XOR	a ^ b	Returns a one in each bit position for which the corresponding bits of either but not both operands are ones.
Bitwise NOT	~ a	Inverts the bits of its operand.
Left shift	a << b	Shifts a in binary representation b bits to left, shifting in zeros from the right.
Sign-propagating right shift	a >> b	Shifts a in binary representation b bits to right, discarding bits shifted off.
Zero-fill right shift	a >>> b	Shifts a in binary representation b bits to the right, discarding bits shifted off, and shifting in zeros from the left.

Implemented in JavaScript 1.0

ECMA version ECMA-262

"> Bitwise Logical Operators

Implemented in	JavaScript 1.0
ECMA version	ECMA-262

Conceptually, the bitwise logical operators work as follows:

-
- The operands are converted to thirty-two-bit integers and expressed by a series of bits (zeros and ones).
- Each bit in the first operand is paired with the corresponding bit in the second operand: first bit to first bit, second bit to second bit, and so on.
- The operator is applied to each pair of bits, and the result is constructed bitwise.

For example, the binary representation of nine is 1001, and the binary representation of fifteen is 1111. So, when the bitwise operators are applied to these values, the results are as follows:

-
- 15 & 9 yields 9 (1111 & 1001 = 1001)
- 15 | 9 yields 15 (1111 | 1001 = 1111)
- 15 ^ 9 yields 6 (1111 ^ 1001 = 0110)

Implemented in	JavaScript 1.0
ECMA version	ECMA-262

"> Bitwise Shift Operators

Implemented in	JavaScript 1.0

ECMA version	ECMA-262

The bitwise shift operators take two operands: the first is a quantity to be shifted, and the second specifies the number of bit positions by which the first operand is to be shifted. The direction of the shift operation is controlled by the operator used.

Shift operators convert their operands to thirty-two-bit integers and return a result of the same type as the left operator.

<< (Left Shift)

This operator shifts the first operand the specified number of bits to the left. Excess bits shifted off to the left are discarded. Zero bits are shifted in from the right.

For example, 9<<2 yields thirty-six, because 1001 shifted two bits to the left becomes 100100, which is thirty-six.

>> (Sign-Propagating Right Shift)

This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Copies of the leftmost bit are shifted in from the left.

For example, 9>>2 yields two, because 1001 shifted two bits to the right becomes 10, which is two. Likewise, -9>>2 yields -3, because the sign is preserved.

>>> (Zero-Fill Right Shift)

This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Zero bits are shifted in from the left.

For example, 19>>>2 yields four, because 10011 shifted two bits to the right becomes 100, which is four. For non-negative numbers, zero-fill right shift and sign-propagating right shift yield the same result.

Logical Operators

Logical operators are typically used with Boolean (logical) values; when they are, they return a Boolean value. However, the && and || operators actually return the value of one of the specified operands, so if these operators are used with non-Boolean values, they may return a non-Boolean value.

Implemented in	JavaScript 1.0
ECMA version	ECMA-262

The logical operators are described in the following table.

Table 5.5 Logical operators

Operator	Usage	Description
&&	*expr1* && *expr2*	(Logical AND) Returns expr1 if it can be converted to false; otherwise, returns expr2. Thus, when used with Boolean values, && returns true if both operands are true; otherwise, returns false.
||	*expr1* || *expr2*	(Logical OR) Returns expr1 if it can be converted to true; otherwise, returns expr2. Thus, when used with Boolean values, || returns true if either operand is true; if both are false, returns false.
!	!*expr*	(Logical NOT) Returns false if its single operand can be converted to true; otherwise, returns true.

Examples of expressions that can be converted to false are those that evaluate to null, 0, the empty string (""), or undefined.

Even though the && and || operators can be used with operands that are not Boolean values, they can still be considered Boolean operators since their return values can always be converted to Boolean values.

Short-Circuit Evaluation. As logical expressions are evaluated left to right, they are tested for possible "short-circuit" evaluation using the following rules:

-
- false && *anything* is short-circuit evaluated to false.
- true || *anything* is short-circuit evaluated to true.

The rules of logic guarantee that these evaluations are always correct. Note that the *anything* part of the above expressions is not evaluated, so any side effects of doing so do not take effect.

Backward Compatibility

JavaScript 1.0 and 1.1. The && and || operators behave as follows:

Operator	Behavior
&&	If the first operand (expr1) can be converted to false, the && operator returns false rather than the value of expr1.
||	If the first operand (expr1) can be converted to true, the || operator returns true rather than the value of expr1.

Examples

The following code shows examples of the && (logical AND) operator.

```
a1=true && true       // t && t returns true
a2=true && false      // t && f returns false
a3=false && true      // f && t returns false
a4=false && (3 == 4)  // f && f returns false
a5="Cat" && "Dog"     // t && t returns Dog
a6=false && "Cat"     // f && t returns false
a7="Cat" && false     // t && f returns false
```

The following code shows examples of the || (logical OR) operator.

```
o1=true || true       // t || t returns true
o2=false || true      // f || t returns true
o3=true || false      // t || f returns true
o4=false || (3 == 4)  // f || f returns false
o5="Cat" || "Dog"     // t || t returns Cat
o6=false || "Cat"     // f || t returns Cat
o7="Cat" || false     // t || f returns Cat
```

The following code shows examples of the ! (logical NOT) operator.

```
n1=!true              // !t returns false
n2=!false             // !f returns true
n3=!"Cat"             // !t returns false
```

String Operators

In addition to the comparison operators, which can be used on string values, the concatenation operator (+) concatenates two string values together, returning another string that is the union of the two operand strings. For example, "my " + "string" returns the string "my string".

Implemented in	JavaScript 1.0
ECMA version	ECMA-262

The shorthand assignment operator += can also be used to concatenate strings. For example, if the variable mystring has the value "alpha," then the expression mystring += "bet" evaluates to "alphabet" and assigns this value to mystring.

Special Operators

?: (Conditional operator)

The conditional operator is the only JavaScript operator that takes three operands. This operator is frequently used as a shortcut for the if statement.

Implemented in	JavaScript 1.0
ECMA version	ECMA-262

Syntax
condition ? *expr1* : *expr2*

Parameters

condition	An expression that evaluates to true or false.

expr1, expr2	Expressions with values of any type.

Description
If condition is true, the operator returns the value of expr1; otherwise, it returns the value of expr2. For example, to display a different message based on the value of the isMember variable, you could use this statement:

document.write ("The fee is " + (isMember ? "$2.00" : "$10.00"))

, (Comma operator)

The comma operator evaluates both of its operands and returns the value of the second operand.

Implemented in	JavaScript 1.0
ECMA version	ECMA-262

Syntax
expr1, expr2

Parameters

expr1, expr2	Any expressions.

Description
You can use the comma operator when you want to include multiple expressions in a location that requires a single expression. The most common usage of this operator is to supply multiple parameters in a for loop.

For example, if a is a 2-dimensional array with 10 elements on a side, the following code uses the comma operator to increment two variables at once. The code prints the values of the diagonal elements in the array:

```
for (var i=0, j=9; i <= 9; i++, j--)
  document.writeln("a["+i+","+j+"]= " + a[i,j])
```

delete

The delete operator deletes an object, an object's property, or an element at a specified index in an array.

Implemented in	JavaScript 1.2, NES 3.0
ECMA version	ECMA-262

Syntax
delete *objectName*
delete *objectName.property*
delete *objectName*[*index*]
delete *property* // legal only within a with statement

Parameters

objectName	The name of an object.
property	The property to delete.
index	An integer representing the array index to delete.

Description
The fourth form is legal only within a with statement, to delete a property from an object.

You can use the delete operator to delete variables declared implicitly but not those declared with the var statement.

If the delete operator succeeds, it sets the property or element to undefined. The delete operator returns true if the operation is possible; it returns false if the operation is not possible.

```
x=42
var y= 43
myobj=new Number()
myobj.h=4      // create property h
delete x       // returns true (can delete if declared implicitly)
delete y       // returns false (cannot delete if declared with var)
delete Math.PI // returns false (cannot delete predefined properties)
delete myobj.h // returns true (can delete user-defined properties)
delete myobj   // returns true (can delete objects)
```

Deleting array elements. When you delete an array element, the array length is not affected. For example, if you delete a[3], a[4] is still a[4] and a[3] is undefined.

When the delete operator removes an array element, that element is no longer in the array. In the following example, trees[3] is removed with delete.

```
trees=new Array("redwood","bay","cedar","oak","maple")
delete trees[3]
if (3 in trees) {
  // this does not get executed
}
```

If you want an array element to exist but have an undefined value, use the undefined keyword instead of the delete operator. In the following example, trees[3] is assigned the value undefined, but the array element still exists:

```
trees=new Array("redwood","bay","cedar","oak","maple")
trees[3]=undefined
if (3 in trees) {
  // this gets executed
}
```

function

The function operator defines an anonymous function inside an expression.

Implemented in	JavaScript 1.5

Syntax

{var | const} *variableName* = function(*parameters*) {*functionBody*};

Description

The following examples shows how the function operator is used.

This example declares an unnamed function inside an expression. It sets x to a function that returns the square of its argument:

```
var x = function(y) {return y*y};
```

The next example declares array a as an array of three functions:

var a = [function(y) {return y}, function y {return y*y}, function (y) [return y*y*y}];

For this array, a[0](5) returns 5, a[1](5) returns 25, and a[2](5) returns 125.

in

The in operator returns true if the specified property is in the specified object.

Implemented in	JavaScript 1.4

Syntax
propNameOrNumber in *objectName*

Parameters

propNameOrNumber	A string or numeric expression representing a property name or array index.
objectName	Name of an object.

Description
The following examples show some uses of the in operator.

// Arrays

```
trees=new Array("redwood","bay","cedar","oak","maple")
0 in trees       // returns true
3 in trees       // returns true
6 in trees       // returns false
"bay" in trees    // returns false (you must specify the index number,
                  // not the value at that index)
"length" in trees // returns true (length is an Array property)

// Predefined objects
"PI" in Math          // returns true
myString=new String("coral")
"length" in myString  // returns true

// Custom objects
mycar = {make:"Honda",model:"Accord",year:1998}
"make" in mycar  // returns true
"model" in mycar // returns true
```

You must specify an object on the right side of the in operator. For example, you can specify a string created with the String constructor, but you cannot specify a string literal.

```
color1=new String("green")
"length" in color1 // returns true
color2="coral"
"length" in color2 // generates an error (color is not a String object)
```

Using in with deleted or undefined properties. If you delete a property with the delete operator, the in operator returns false for that property.

```
mycar = {make:"Honda",model:"Accord",year:1998}
delete mycar.make
"make" in mycar  // returns false

trees=new Array("redwood","bay","cedar","oak","maple")
delete trees[3]
3 in trees // returns false
```

If you set a property to undefined but do not delete it, the in operator returns true for that property.

```
mycar = {make:"Honda",model:"Accord",year:1998}
mycar.make=undefined
```

```
"make" in mycar  // returns true

trees=new Array("redwood","bay","cedar","oak","maple")
trees[3]=undefined
3 in trees // returns true
```

For additional information about using the in operator with deleted array elements, see delete.

instanceof

The instanceof operator returns true if the specified object is of the specified object type.

Implemented in	JavaScript 1.4

Syntax

objectName instanceof *objectType*

Parameters

objectName	Name of the object to compare to objectType.
objectType	Object type.

Description

Use instanceof when you need to confirm the type of an object at runtime. For example, when catching exceptions, you can branch to different exception-handling code depending on the type of exception thrown.

You must specify an object on the right side of the instanceof operator. For example, you can specify a string created with the String constructor, but you cannot specify a string literal.

```
color1=new String("green")
color1 instanceof String // returns true
color2="coral"
color2 instanceof String // returns false (color is not a String object)
```

Examples

See also the examples for throw.

Example 1. The following code uses instanceof to determine whether theDay is a Date object. Because theDay is a Date object, the statements in the if statement execute.

```
theDay=new Date(1995, 12, 17)
if (theDay instanceof Date) {
  // statements to execute
}
```

Example 2. The following code uses instanceof to demonstrate that String and Date objects are also of type Object (they are derived from Object).

```
myString=new String()
myDate=new Date()

myString instanceof String // returns true
myString instanceof Object // returns true
myString instanceof Date   // returns false

myDate instanceof Date     // returns true
myDate instanceof Object   // returns true
myDate instanceof String   // returns false
```

Example 3. The following code creates an object type Car and an instance of that object type, mycar. The instanceof operator demonstrates that the mycar object is of type Car and of type Object.

```
function Car(make, model, year) {
  this.make = make
  this.model = model
  this.year = year
}
mycar = new Car("Honda", "Accord", 1998)
a=mycar instanceof Car // returns true
b=mycar instanceof Object // returns true
```

new

The new operator creates an instance of a user-defined object type or of one of the built-in object types that has a constructor function.

Implemented in	JavaScript 1.0
ECMA version	ECMA-262

Syntax

objectName = new *objectType* (*param1* [,*param2*] ...[,*paramN*])

Parameters

objectName	Name of the new object instance.

objectType	Object type. It must be a function that defines an object type.
param1...paramN	Property values for the object. These properties are parameters defined for the objectType function.

Description
Creating a user-defined object type requires two steps:

1.
1. Define the object type by writing a function.

2. Create an instance of the object with new.

To define an object type, create a function for the object type that specifies its name, properties, and methods. An object can have a property that is itself another object. See the examples below.

You can always add a property to a previously defined object. For example, the statement car1.color = "black" adds a property color to car1, and assigns it a value of "black". However, this does not affect any other objects. To add the new property to all objects of the same type, you must add the property to the definition of the car object type.

You can add a property to a previously defined object type by using the Function.prototype property. This defines a property that is shared by all objects created with that function, rather than by just one instance of the object type. The following code adds a color property to all objects of type car, and then assigns a value to the color property of the object car1. For more information, see prototype

Car.prototype.color=null
car1.color="black"
birthday.description="The day you were born"

Examples
Example 1: Object type and object instance. Suppose you want to create an object type for cars. You want this type of object to be called car, and you want it to have

properties for make, model, and year. To do this, you would write the following function:

```
function car(make, model, year) {
  this.make = make
  this.model = model
  this.year = year
}
```

Now you can create an object called mycar as follows:

```
mycar = new car("Eagle", "Talon TSi", 1993)
```

This statement creates mycar and assigns it the specified values for its properties. Then the value of mycar.make is the string "Eagle", mycar.year is the integer 1993, and so on.

You can create any number of car objects by calls to new. For example,

```
kenscar = new car("Nissan", "300ZX", 1992)
```

Example 2: Object property that is itself another object. Suppose you define an object called person as follows:

```
function person(name, age, sex) {
  this.name = name
  this.age = age
  this.sex = sex
}
```

And then instantiate two new person objects as follows:

```
rand = new person("Rand McNally", 33, "M")
ken = new person("Ken Jones", 39, "M")
```

Then you can rewrite the definition of car to include an owner property that takes a person object, as follows:

```
function car(make, model, year, owner) {
  this.make = make;
  this.model = model;
  this.year = year;
  this.owner = owner;
}
```

To instantiate the new objects, you then use the following:

```
car1 = new car("Eagle", "Talon TSi", 1993, rand);
car2 = new car("Nissan", "300ZX", 1992, ken)
```

Instead of passing a literal string or integer value when creating the new objects, the above statements pass the objects rand and ken as the parameters for the owners. To find out the name of the owner of car2, you can access the following property:

```
car2.owner.name
```

this

The this keyword refers to the current object. In general, in a method this refers to the calling object.

Implemented in	JavaScript 1.0
ECMA version	ECMA-262

Syntax
this[.*propertyName*]

Examples
Suppose a function called validate validates an object's value property, given the object and the high and low values:

```
function validate(obj, lowval, hival) {
  if ((obj.value < lowval) || (obj.value > hival))
    alert("Invalid Value!")
}
```

You could call validate in each form element's onChange event handler, using this to pass it the form element, as in the following example:

```
<B>Enter a number between 18 and 99:</B>
<INPUT TYPE = "text" NAME = "age" SIZE = 3
  onChange="validate(this, 18, 99)">
```

typeof

The typeof operator is used in either of the following ways:

1. typeof *operand*
2. typeof (*operand*)

The typeof operator returns a string indicating the type of the unevaluated operand. operand is the string, variable, keyword, or object for which the type is to be returned. The parentheses are optional.

Implemented in	JavaScript 1.1
ECMA version	ECMA-262

Suppose you define the following variables:

```
var myFun = new Function("5+2")
var shape="round"
var size=1
var today=new Date()
```

The typeof operator returns the following results for these variables:

```
typeof myFun is object
typeof shape is string
typeof size is number
typeof today is object
typeof dontExist is undefined
```

For the keywords true and null, the typeof operator returns the following results:

```
typeof true is boolean
```

typeof null is object

For a number or string, the typeof operator returns the following results:

typeof 62 is number
typeof 'Hello world' is string

For property values, the typeof operator returns the type of value the property contains:

typeof document.lastModified is string
typeof window.length is number
typeof Math.LN2 is number

For methods and functions, the typeof operator returns results as follows:

typeof blur is function
typeof eval is function
typeof parseInt is function
typeof shape.split is function

For predefined objects, the typeof operator returns results as follows:

typeof Date is function
typeof Function is function
typeof Math is function
typeof Option is function
typeof String is function

void

The void operator is used in either of the following ways:

1. void (*expression*)
2. void *expression*

The void operator specifies an expression to be evaluated without returning a value. expression is a JavaScript expression to evaluate. The parentheses surrounding the expression are optional, but it is good style to use them.

Implemented in	JavaScript 1.1
ECMA version	ECMA-262

You can use the void operator to specify an expression as a hypertext link. The expression is evaluated but is not loaded in place of the current document.

The following code creates a hypertext link that does nothing when the user clicks it. When the user clicks the link, void(0) evaluates to 0, but that has no effect in JavaScript.

```
<A HREF="javascript:void(0)">Click here to do nothing</A>
```

The following code creates a hypertext link that submits a form when the user clicks it.

```
<A HREF="javascript:void(document.form.submit())">
Click here to submit</A>
```

Previous **Contents** **Index** **Next**

Last Updated **September 28, 2000**

Part 3 LiveConnect Class Reference

Chapter 6 Java Classes, Constructors, and Methods
This chapter documents the Java classes used for LiveConnect, along with their constructors and methods. It is an alphabetical reference for the classes that allow a Java object to access JavaScript code.

Last Updated **September 28, 2000**

Chapter 6 Chapter 6 Java Classes, Constructors, and Methods

This chapter documents the Java classes used for LiveConnect, along with their constructors and methods. It is an alphabetical reference for the classes that allow a Java object to access JavaScript code.

This reference is organized as follows:

-
- Full entries for each class appear in alphabetical order.
  Tables included in the description of each class summarize the constructors and methods of the class.
- Full entries for the constructors and methods of a class appear in alphabetical order after the entry for the class.

Last Updated **September 28, 2000**

JSException

The public class JSException extends RuntimeException.

```
java.lang.Object
  |
  +----java.lang.Throwable
         |
         +----java.lang.Exception
                |
                +----java.lang.RuntimeException
                       |
                       +----netscape.javascript.JSException
```

Description

JSException is an exception which is thrown when JavaScript code returns an error.

Constructor Summary

The netscape.javascript.JSException class has the following constructors:

Constructor	Description
JSException	Deprecated constructors optionally let you specify a detail message and other information.

Method Summary

The netscape.javascript.JSException class has the following methods:

Method	Description
getWrappedException	Instance method getWrappedException.
getWrappedExceptionType	Instance method getWrappedExceptionType returns the int mapping of the type of the wrappedException object.

The following sections show the declaration and usage of the constructors and method.

Backward Compatibility

JavaScript 1.1 through 1.3. JSException had three public constructors which optionally took a string argument, specifying the detail message or other information for the exception. The getWrappedException method was not available.

JSException

Constructors, deprecated in JavaScript 1.4. Constructs a JSException with an optional detail message.

Declaration

1. public JSException()

2. public JSException(String s)

3. public JSException(String s,
   String filename,
   int lineno,
   String source,

int tokenIndex)

Arguments

s	The detail message.
filename	The URL of the file where the error occurred, if possible.
lineno	The line number if the file, if possible.
source	The string containing the JavaScript code being evaluated.
tokenIndex	The index into the source string where the error occurred.

getWrappedException

Instance method getWrappedException.

Declaration

public Object getWrappedException()

Description

getWrappedException() returns an object that represents the value that the JavaScript actually threw. JavaScript can throw any type of value. Use getWrappedException() to determine what kind of value the Object return type represents.

getWrappedExceptionType

Instance method getWrappedExceptionType.

Declaration
public int getWrappedExceptionType()

Description
getWrappedExceptionType() returns an int that matches one of the following static ints declared by the JSException class:

EXCEPTION_TYPE_EMPTY
EXCEPTION_TYPE_VOID
EXCEPTION_TYPE_OBJECT
EXCEPTION_TYPE_FUNCTION
EXCEPTION_TYPE_STRING
EXCEPTION_TYPE_NUMBER
EXCEPTION_TYPE_BOOLEAN

Previous **Contents** **Index** **Next**

Last Updated **September 28, 2000**

JSObject

The public final class netscape.javascript.JSObject extends Object.

```
java.lang.Object
  |
  +----netscape.javascript.JSObject
```

Description

JavaScript objects are wrapped in an instance of the class netscape.javascript.JSObject and passed to Java. JSObject allows Java to manipulate JavaScript objects.

When a JavaScript object is sent to Java, the runtime engine creates a Java wrapper of type JSObject; when a JSObject is sent from Java to JavaScript, the runtime engine unwraps it to its original JavaScript object type. The JSObject class provides a way to invoke JavaScript methods and examine JavaScript properties.

Any JavaScript data brought into Java is converted to Java data types. When the JSObject is passed back to JavaScript, the object is unwrapped and can be used by JavaScript code. See the *Core JavaScript Guide* for more information about data type conversions.

Method Summary

The netscape.javascript.JSObject class has the following methods:

Method	Description

call	Calls a JavaScript method.
equals	Determines if two JSObject objects refer to the same instance.
eval	Evaluates a JavaScript expression.
getMember	Retrieves the value of a property of a JavaScript object.
getSlot	Retrieves the value of an array element of a JavaScript object.
removeMember	Removes a property of a JavaScript object.
setMember	Sets the value of a property of a JavaScript object.
setSlot	Sets the value of an array element of a JavaScript object.

toString	Converts a JSObject to a string.

The netscape.javascript.JSObject class has the following static methods:

Method	Description
getWindow	Gets a JSObject for the window containing the given applet.

The following sections show the declaration and usage of these methods.

call

Method. Calls a JavaScript method. Equivalent to "this.methodName(args[0], args[1], ...)" in JavaScript.

Declaration
public Object call(String methodName,
  Object args[])

equals

Method. Determines if two JSObject objects refer to the same instance.

Overrides: equals in class java.lang.Object

Declaration
public boolean equals(Object obj)

Backward Compatibility

JavaScript 1.3. In JavaScript 1.3 and earlier versions, you can use either the equals method of java.lang.Object or the == operator to evaluate two JSObject objects.

In more recent versions, the same JSObject can appear as different Java objects. You can use the equals method to determine whether two JSObjects refer to the same instance.

eval

Method. Evaluates a JavaScript expression. The expression is a string of JavaScript source code which will be evaluated in the context given by "this".

Declaration
public Object eval(String s)

getMember

Method. Retrieves the value of a property of a JavaScript object. Equivalent to "this.name" in JavaScript.

Declaration
public Object getMember(String name)

getSlot

Method. Retrieves the value of an array element of a JavaScript object. Equivalent to "this[index]" in JavaScript.

Declaration
public Object getSlot(int index)

getWindow

Static method. Returns a JSObject for the window containing the given applet. This method is useful in client-side JavaScript only.

Declaration
public static JSObject getWindow(Applet applet)

removeMember

Method. Removes a property of a JavaScript object.

Declaration
public void removeMember(String name)

setMember

Method. Sets the value of a property of a JavaScript object. Equivalent to "this.name = value" in JavaScript.

Declaration
public void setMember(String name,
  Object value)

setSlot

Method. Sets the value of an array element of a JavaScript object. Equivalent to "this[index] = value" in JavaScript.

Declaration
public void setSlot(int index,
  Object value)

toString

Method. Converts a JSObject to a String.

Overrides: toString in class java.lang.Object

Declaration
public String toString()

Previous **Contents** **Index** **Next**

Last Updated **September 28, 2000**

Part 4 Appendixes

Last Updated **September 28, 2000**

Appendix A Appendix A Reserved Words

This appendix lists the reserved words in JavaScript.

The reserved words in this list cannot be used as JavaScript variables, functions, methods, or object names. Some of these words are keywords used in JavaScript; others are reserved for future use.

abstract	else	instanceof	switch
boolean	enum	int	synchronized
break	export	interface	this
byte	extends	long	throw
case	false	native	throws
catch	final	new	transient
char	finally	null	true
class	float	package	try
const	for	private	typeof
continue	function	protected	var
debugger	goto	public	void
default	if	return	volatile
delete	implements	short	while
do	import	static	with
double	in	super	

Last Updated **September 28, 2000**

Appendix B Appendix B Deprecated Features

This appendix lists the features that are deprecated as of JavaScript 15.

-
- RegExp Properties

  The following properties are deprecated.

Property	Description
$1, ..., $9	Parenthesized substring matches, if any.
$_	See input.
$*	See multiline.
$&	See lastMatch.

$+	See lastParen.
$`	See leftContext.
$'	See rightContext.
input	The string against which a regular expression is matched.
lastMatch	The last matched characters.
lastParen	The last parenthesized substring match, if any.
leftContext	The substring preceding the most recent match.
rightContext	The substring following the most recent match.

The following are now properties of RegExp instances, no longer of the RegExp object.

Property	Description
global	Whether or not to test the regular expression against all possible matches in a string, or only against the first.
ignoreCase	Whether or not to ignore case while attempting a match in a string.
lastIndex	The index at which to start the next match.
multiline	Whether or not to search in strings across multiple lines.
source	The text of the pattern.

- RegExp Methods
  The compile method is deprecated.
  The valueOf method is no longer specialized for RegExp. Use Object.valueOf.
- Escape sequences
  Octal escape sequences (\ followed by one, two, or three octal digits) are deprecated in string and regular expression literals.
  The escape and unescape functions are deprecated. Use encodeURI, encodeURIComponent, decodeURI or decodeURIComponent to encode and decode escape sequences for special characters.

Previous **Contents** **Index** **Next**

Last Updated **September 28, 2000**

Symbols A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

Previous **Contents**

Last Updated **September 28, 2000**