• Client side and server side java script

Client-side JavaScript is embedded directly in HTML pages and is interpreted by the browser completely at runtime.

Web browsers such can interpret client-side JavaScript statements embedded in an HTML page. When the browser (or client) requests such a page, the server sends the full content of the document, including HTML and JavaScript statements, over the network to the client. The client reads the page from top to bottom, displaying the results of the HTML and executing JavaScript statements as it goes

Client-side JavaScript statements embedded in an HTML page can respond to user events such as mouse clicks, form input, and page navigation. For example, you can write a JavaScript function to verify that users enter valid information into a form requesting a telephone number or zip code. Without any network transmission, the embedded JavaScript on the HTML page can check the entered data and display a dialog box to the user who enters invalid data.

On the server, JavaScript is also embedded in HTML pages. The server-side statements can connect to relational databases from different vendors, share information across users of an application, access the file system on the server, or communicate with other applications through LiveConnect and Java. A compiled JavaScript application can also include client-side JavaScript in addition to server-side JavaScript.

In contrast to pure client-side JavaScript scripts, JavaScript applications that use server-side JavaScript are compiled into bytecode executable files. These application executables are run in concert with a web server that contains the JavaScript runtime engine. For this reason, creating JavaScript applications is a two-stage process.

In the first stage you (the developer) create HTML pages (which can contain both client-side and server-side JavaScript statements) and JavaScript files. You then compile all of those files into a single executable.

In the second stage, a page in the application is requested by a client browser. The runtime engine uses the application executable to look up the source page and dynamically generate the HTML page to return. It runs any server-side JavaScript statements found on the page. The result of those statements might add new HTML or client-side JavaScript statements to the HTML page. It then sends the resulting page over the network to the Navigator client, which displays the results.

• Java Script Objects

JavaScript has predefined objects for the core language, as well as additions for client-side and server-side JavaScript.

JavaScript has the following core objects:

Array, Boolean, Date, Function, Math, Number, Object, String

The additional client-side objects are as follows:

Anchor, Applet, Area, Button, Checkbox, document, event, FileUpload, Form, Frame, Hidden, History, Image, Layer, Link, Location, MimeType, navigator, Option, Password, Plugin, Radio, Reset, screen, Select, Submit, Text, Textarea, Window

These objects represent information relevant to working with JavaScript in a web browser. Many of these objects are related to each other by occurring as property values. For example, to access the images in a document, you use the document.images array, each of whose elements is a Image object. Figure 1.5 shows the client-side object containment hierarchy.

- ## JavaScript Security

Navigator version 2.02 and later automatically prevents scripts on one server from accessing properties of documents on a different server. This restriction prevents scripts from fetching private information such as directory structures or user session history.

JavaScript for Navigator 3.0 has a feature called data tainting that retains the security restriction but provides a means of secure access to specific components on a page.

When data tainting is enabled, JavaScript in one window can see properties of another window, no matter what server the other window's document was loaded from. However, the author of the other window *taints* (marks) property values or other data that should be secure or private, and JavaScript cannot pass these tainted values on to any server without the user's permission.

When data tainting is disabled, a script cannot access any properties of a window on another server.

In Navigator 4.0, data tainting has been removed. Instead, Navigator 4.0 provides signed JavaScript scripts for more reliable and more flexible security

- ## JavaScript operators

JavaScript has assignment, comparison, arithmetic, bitwise, logical, string, and special operators.

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 true if both logical operands are true. Otherwise, returns false.
	||	(Logical OR) Returns true if either logical expression is true. If both are false, returns false.
	!	(Logical negation) If its single operand is true, returns false; 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	==	Returns true if the operands are equal.

Operators	!=	Returns true if the operands are not equal.
	>	Returns true if left operand is greater than right operand.
	>=	Returns true if left operand is greater than or equal to right operand.
	<	Returns true if left operand is less than right operand.
	<=	Returns true if left operand is less than or equal to right operand.
Special Operators	?:	Lets you perform a simple **"if...then...else"**
	,	Evaluates two expressions and returns the result of the second expression.
	delete	Lets you delete an object property or an element at a specified index in an array.
	new	Lets you create 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	The void operator specifies an expression to be evaluated without returning a value.

Special Operators

1. ?: (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.

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"))
```

2. , (Comma operator)

The comma operator is very simple. It evaluates both of its operands and returns the value of the second operand.

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=10; i <= 10; i++, j--)
  document.writeln("a["+i+","+j+"]= " + a[i,j])
```

3. delete

Deletes an object's property or an element at a specified index in an array.

Syntax

```
delete objectName.property
delete objectName[index]
delete property
```

Parameters

objectName	The name of an object.
property	An existing property.
Index	An integer representing the location of an element in an array

Description

The third form is legal only within a **with** statement.

If the deletion succeeds, the **delete** operator sets the property or element to **undefined**. **delete** always returns undefined.

4. new

An operator that lets you create an instance of a user-defined object type or of one of the built-in object types that has a constructor function.

Syntax

objectName = new objectType (param1 [,param2] ...[,paramN])

Arguments

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:

Define the object type by writing a function.

Create an instance of the object with **new**.

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)
```

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)
```

5. this

A keyword that you can use to refer to the current object. In general, in a method **this** refers to the calling object.

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)">
```

6. 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.

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

7. void

The void operator is used in either of the following ways:

1. javascript:void (expression)
2. javascript: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.

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>
```

• Statements in javaScript

1. break

Terminates the current while or for loop and transfers program control to the statement following the terminated loop.

Syntax

```
break
break label
```

Argument

label Identifier associated with the label of the statement.

Description

The break statement can now include an optional label that allows the program to break out of a labeled statement. This type of break must be in a statement identified by the label used by break.

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
}
```

2. comment

Notations by the author to explain what a script does. Comments are ignored by the interpreter.

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. */
```

3. delete

Deletes an object's property or an element at a specified index in an array.

Syntax

```
delete objectName.property
delete objectName[index]
delete property
```

Arguments

objectName	An object from which to delete the specified property or value.
property	The property to delete.
index	An integer index into an array.

Description
If the delete operator succeeds, it sets the property of element to undefined; the operator always returns undefined.

You can only use the delete operator to delete object properties and array entries. You cannot use this operator to delete objects or variables. Consequently, you can only use the third form within a with statement, to delete a property from the object.

4. do...while

Executes its statements until the test condition evaluates to false. Statement is executed at least once.

Syntax
```
do
  statement
while (condition);
```

Arguments

statement	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.

Example
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);
```

5. export

Allows a signed script to provide properties, functions, and objects to other signed or unsigned scripts.

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.

6. 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.

Syntax

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

Arguments

initial-expression	Statement or variable declaration. Typically used to initialize a counter variable. This expression may optionally declare new variables with the var keyword.
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)
}
```

7. for...in

Iterates a specified variable over all the properties of an object. For each distinct property, JavaScript executes the specified statements.

Syntax

```
for (variable in object) {
  statements}
```

Arguments

variable	Variable to iterate over every property.
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 dump_props(obj, objName) {
  var result = ""
  for (var i in obj) {
    result += objName + "." + i + " = " + obj[i] + "<BR>"
  }
  result += "<HR>"
  return result
}
```

8. function

Declares a JavaScript function with the specified parameters. Acceptable parameters include strings, numbers, and objects.

Syntax

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

Arguments

name	The function name.
param	The name of an argument to be passed to the function. A function can have up to 255 arguments.

Description

To return a value, the function must have a return statement that specifies the value to return. You cannot nest a function statement in another statement or in itself.

All parameters are passed to functions, *by value*. In other words, 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.

In addition to defining functions as described here, you can also define Function objects.

Examples

```
//This function 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
}
```

9. 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.

Syntax

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

Arguments

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.

Examples

```
if (cipher_char == from_char) {
   result = result + to_char
   x++}
else
   result = result + clear_char
```

10. import

Allows a script to import properties, functions, and objects from a signed script which has exported the information.

Syntax

```
import objectName.name1, objectName.name2, ..., objectName.nameN
import objectName.*
```

Parameters

nameN	List of properties, functions, and objects to import from the export file.
objectName	Name of the object that will receive the imported names.

*	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, then

import obj.f, obj.p
makes f and p accessible in the importing script as properties of obj.

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.

11. labeled
Provides an identifier that can be used with break or continue to indicate where the program should continue execution.

In a labeled statement, break or continue must be followed with a label, and the label must be the identifier of the labeled statement containing break or continue.

Syntax
```
label :
   statement
```

Arguments

statement	Block of statements. break can be used with any labeled statement, and continue can be used with looping labeled statements.

Example
For an example of a labeled statement using break, see break. For an example of a labeled statement using continue, see continue.

12. return
Specifies the value to be returned by a function.

Syntax
```
return expression
```

Examples
The following function returns the square of its argument, x, where x is a number.

```
function square(x) {
  return x * x
}
```

13. switch

Allows a program to evaluate an expression and attempt to match the expression's value to a case label.

Syntax

```
switch (expression){
  case label :
    statement;
    break;
  case label :
    statement;
    break;
  ...
  default : statement;
}
```

Arguments

expression	Value matched against label.
label	Identifier used to match against expression.
statement	Any statement.

Description

If a match is found, the program executes the associated statement.

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.

Example

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>");
```

14. var

Declares a variable, optionally initializing it to a value.

Syntax

```
var varname [= value] [..., varname [= value] ]
```

Arguments

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; 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 if a global variable of the same name exists.

Examples

```
var num_hits = 0, cust_no = 0
```

15. 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.

Syntax

```
while (condition) {
  statements
}
```

Arguments

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.

16. with

Establishes the default object for a set of statements. Within the set of statements, any property references that do not specify an object are assumed to be for the default object.

Syntax

```
with (object){
  statements
}
```

Arguments

object	Specifies the default object to use for the statements. The parentheses around object are required.
statements	Any block of statements.

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)
}
```

• Arrays

```
new Array(arrayLength);
new Array(element0, element1, ..., elementN);
```

Parameters

arrayLength	(Optional) The initial length of the array. You can access this value using the length property.
element*N*	(Optional) 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

In Navigator 3.0, you can specify an initial length when you create the array. The following code creates an array of five elements:

```
billingMethod = new Array(5)
```

When you create an array, all of its elements are initially null. The following code creates an array of 25 elements, then assigns values to the first three elements:

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

Method Summary

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 that last element added.
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.
toString	Returns a string representing the specified object.
unshift	Adds one or more elements to the front of an array and returns the new length of the array.

Give Example taken in class

• Boolean

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

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, null, false, or the empty string (""), the object has an initial value of false. All other values, including the string "false", create an object with an initial value of true.

Description
Use a Boolean object when you need to convert a non-boolean value to a boolean value. You can use the Boolean object any place JavaScript expects a primitive boolean value. JavaScript returns the primitive value of the Boolean object by automatically invoking the valueOf method.

Property Summary

prototype Defines a property that is shared by all Boolean objects.

Method Summary

toString Returns a string representing the specified 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")
```

Properties

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

Methods

toString

Returns a string representing the specified object.

Method of Boolean

Syntax
toString()

Parameters
None.

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.

You can use toString within your own code to convert an object into a string, and you can create your own function to be called in place of the default toString method.

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

```
flag = new Boolean(true)
document.write("flag.toString() is " + flag.toString() + "<BR>")
```

- ## Date

Lets you work with dates and times.

Created by

The Date constructor:

```
new Date()
new Date("month day, year hours:minutes:seconds")
new Date(yr_num, mo_num, day_num)
new Date(yr_num, mo_num, day_num, hr_num, min_num, sec_num)
```

Parameters

month, day, year, hours, minutes, seconds	String values representing part of a date.
yr_num, mo_num, day_num, hr_num, min_num, sec_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.

Description

If you supply no arguments, the constructor creates a Date object for today's date and 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, and seconds.

The way JavaScript handles dates is very similar to the way Java handles dates: both languages have many of the same date methods, and both store dates internally as the number of milliseconds since January 1, 1970 00:00:00. Dates prior to 1970 are not allowed.

Property Summary

prototype Allows the addition of properties to a Date object.

Method Summary

getDate	Returns the day of the month for the specified date.
getDay	Returns the day of the week for the specified date.
getHours	Returns the hour in the specified date.
getMinutes	Returns the minutes in the specified date.
getMonth	Returns the month in the specified date.

getSeconds	Returns the seconds in the specified date.
getTime	Returns the numeric value corresponding to the time for the specified date.
getTimezoneOffset	Returns the time-zone offset in minutes for the current locale.
getYear	Returns the year in the specified date.
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.
setHours	Sets the hours for a specified date.
setMinutes	Sets the minutes for a specified date.
setMonth	Sets the month for a specified date.
setSeconds	Sets the seconds for a specified date.
setTime	Sets the value of a Date object.
setYear	Sets the year for a specified date.
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.
UTC	Returns the number of milliseconds in a Date object since January 1, 1970, 00:00:00, Universal Coordinated Time (GMT).

For Example

```
<html><head><title>Date Object</title></head>

<body><script language="JavaScript">

{

        currentdate = new Date();

        with(currentdate)

        {

                document.write("Date: " + getDate() + "/" + getMonth() + "/" + getYear() +
"<br>");
```

```
                    document.write("Time: " + getHours() + ":" + getMinutes() + ":" +
getSeconds() + ":" +

getMilliseconds()+"<BR>");

                    document.write("Day of the week :" + getDay()+"<BR>");

                    document.write("Time : " + getTime()+"<BR>");

          }

}</script></body></html>
```

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

- ***Created by***
- The `Function` constructor:
- `new Function (arg1, arg2, ... argN, functionBody)`
-
- ***Parameters***

`arg1, arg2, ... argN`	(Optional) 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 `"theForm"`.
`functionBody`	A string containing the JavaScript statements comprising the function definition.

The following code specifies a `Function` object that takes two arguments.

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

arity

When the `LANGUAGE` attribute of the `SCRIPT` tag is "JavaScript1.2", this property indicates the number of arguments expected by a function.

Property of `Function`

Description

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

Example

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

```
<SCRIPT LANGUAGE = "JavaScript1.2">
function addNumbers(x,y){
   document.write("length = " + arguments.length + "<BR>")
   z = x + y
}
document.write("arity = " + addNumbers.arity + "<BR>")
addNumbers(3,4,5)
</SCRIPT>
```

This script writes:

```
arity = 2
length = 3
```

caller

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

Property of `Function`

Description

The `caller` property is available only within the body of a function. If used outside a function declaration, the `caller` property is null.

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

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	`Object`

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 new 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"
```

• 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.

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

E	Euler's constant and the base of natural logarithms, approximately 2.718.
LN10	Natural logarithm of 10, approximately 2.302.
LN2	Natural logarithm of 2, approximately 0.693.
LOG10E	Base 10 logarithm of E (approximately 0.434).
LOG2E	Base 2 logarithm of E (approximately 1.442).
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

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.

For Example

```
<html><head><title>Math object</title></head>
<body><SCRIPT LANGUAGE="JavaScript">
{
        document.write("<br>" + Math.PI + "<BR>" + Math.E);
        document.write("<p>" + Math.ceil(1.782) + "<BR>" + Math.ceil(1.234));
        document.write("<p>" + Math.floor(1.782) + "<BR>" + Math.floor(1.234));
        document.write("<p>" + Math.abs(-13.77) + "<BR>" + Math.abs(13.77));
        document.write("<p>" + Math.round(13.77) + "<br>" + Math.round(13.10));
        document.write("<p>" + Math.max(13,77));
        document.write("<p>" + Math.min(13,77));
        document.write("<p>" + Math.pow(13,2) + "<br>" + Math.pow(2,10));
        document.write("<p>" + Math.sqrt(9));
        document.write("<p>" + Math.random());
}</script></body></html>
```

- **Number**

Lets you work with numeric values. The `Number` object is an object wrapper for primitive numeric values.

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.

Property

`MAX_VALUE`	The largest representable number.
`MIN_VALUE`	The smallest representable number.
`NaN`	Special "not a number" value.
`NEGATIVE_INFINITY`	Special infinite value; returned on overflow.
`POSITIVE_INFINITY`	Special negative infinite value; returned on overflow.
`prototype`	Allows the addition of properties to a `Number` object.

Method

`toString`	Returns a string representing the specified object.

• 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

Created by
The Object constructor:

new Object();

Method Summary

eval	Evaluates a string of JavaScript code in the context of the specified 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.

For Example

```
<html>

<head>
```

```
</head>

<body>

<script type="text/javascript">

emp=new employee();

emp.name="shital";

emp.age=37;

document.write(emp.name);

document.write(emp.age);

</script>

</body>
```

- ## 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

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 (Optional) If specified, flags can have one of the following 3 values:

- g: global match

- i: ignore case
- gi: both global match and ignore case

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+/
```

Table 4.3 provides a complete list and description of the special characters that can be used in regular expressions.

Table 4.3 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 or line. For example, /^A/ does not match the 'A' in "an A," but does match it in "An A."
$	Matches end of input or line. For example, /t$/ does not match the 't' in "eater", but does match it in "eat"
*	Matches the preceding character 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 character 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 character 0 or 1 time. For example, /e?le?/ matches the 'el' in "angel" and the 'le' in "angle."
.	(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. 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|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 character. 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 character. 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 character. 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 control character. 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\v]. for example, /\s\w*/ matches ' bar' in "foo bar."
\S	Matches a single character other than white space. Equivalent to [^ \f\n\r\t\v]. 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. **Note:** If the number of left parentheses is less than the number specified in \n, the \n is taken as an octal escape as described in the next row.
\o*octal* \x*hex*	Where \o*octal* is an octal escape value or \x*hex* is a hexadecimal escape value. Allows you to embed ASCII codes into regular expressions.

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 expressionobject, 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. Once you have a defined regular expression, and if the regular expression is used throughout the script and may change, you can use the compile method to compile a new regular expression for efficient reuse.

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.

The predefined RegExp object contains the static properties input, multiline, lastMatch, lastParen, leftContext, rightContext, and $1 through $9. The input and multiline properties can be preset. The values for the other static properties are set after execution of the exec and test methods of an individual regular expression object, and after execution of the match and replace methods of String.

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.

$1, ..., $9	Parenthesized substring matches, if any.
$_	See input.
$*	See multiline.
$&	See lastMatch.
$+	See lastParen.
$`	See leftContext.
$'	See rightContext.
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.
input	The string against which a regular expression is matched.
lastIndex	The index at which to start the next match.
lastMatch	The last matched characters.
lastParen	The last parenthesized substring match, if any.
leftContext	The substring preceding the most recent match.
multiline	Whether or not to search in strings across multiple lines.
rightContext	The substring following the most recent match.
source	The text of the pattern.

• String
An object representing a series of characters in a string.

Core object

Created by
The String constructor:

new String(string);

Parameters

string	Any string.

Description
The String object is a built-in JavaScript object. You an treat any JavaScript string as a String object.

A string can be represented as a literal enclosed by single or double quotation marks; for example, "Netscape" or 'Netscape'.

Property Summary

length	Reflects the length of the string.
prototype	Allows the addition of properties to a String object.

Method Summary

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 ISO-Latin-1 codeset 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 from the specified sequence of numbers that are ISO-Latin-1 codeset values.
indexOf	Returns the index within the calling String object of the first occurrence of the specified value.
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.
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.
toUpperCase	Returns the calling string value converted to uppercase.

For Example

1 .

```
<html><head><title>String Object</title></head>
<body><script type="text/javascript">
{
        s = new String("This is a test of the JavaScript String methods.")
        document.write('s = ' + s +"<br>");
        document.write(s.bold()+"<br>");
        document.write(s.small()+"<br>");
        document.write(s.strike()+"<br>");
        document.write(s.big()+"<br>");
        document.write(s.blink()+"<br>");
s}</script></body></html>
```

2

```
<html><head><title>String Object</title></head>
<body><script type="text/javascript">
{
        s = new String("This is a test of the JavaScript String methods.")
        document.write('s = ' + s +"<br>");
        document.write('s.charAt(1) = ' + s.charAt(1)+"<br>");
        document.write('s.charCodeAt(1) = ' + s.charCodeAt(1)+"<br>");
        document.write('s.indexOf("is") = ' + s.indexOf("is")+"<br>");
        document.write('s.lastIndexOf("is") = ' + s.lastIndexOf("is")+"<br>");
        document.write('s.substring(22,32) = ' + s.substring(22,32)+"<br>");
        document.write('s.toLowerCase() = ' + s.toLowerCase()+"<br>");
        document.write('s.toUpperCase() = ' + s.toUpperCase()+"<br>");
        split = s.split(" ");
```

```
                        for(i=0; i<split.length; ++i)

                        {

                                document.write('split[' + i + '] = ' + split[i]+"<br>");

                        }

                }</script></body></html>
```

• Anchor

A place in a document that is the target of a hypertext link.

Created by
Using the HTML A tag or calling the String.anchor method. The JavaScript runtime engine creates an Anchor object corresponding to each A tag in your document that supplies the NAME attribute. It puts these objects in an array in the document.anchors property. You access an Anchor object by indexing this array.

To define an anchor with the String.anchor method:

```
theString.anchor(nameAttribute)
```
where:

theString	A String object.
nameAttribute	A string.

To define an anchor with the A tag, use standard HTML syntax. If you specify the NAME attribute, you can use the value of that attribute to index into the anchors array.

Description
If an Anchor object is also a Link object, the object has entries in both the anchors and links arrays.

Properties
None.

Methods
None.

Examples
Example 1: An anchor. The following example defines an anchor for the text "Welcome to JavaScript":

```
<A NAME="javascript_intro"><H2>Welcome to JavaScript</H2></A>
```

If the preceding anchor is in a file called intro.html, a link in another file could define a jump to the anchor as follows:

```
<A HREF="intro.html#javascript_intro">Introduction</A>
```

Example 2: anchors array. The following example opens two windows. The first window contains a series of buttons that set location.hash in the second window to a specific anchor. The second window defines four anchors named "0," "1," "2," and "3." (The anchor names in the document are therefore 0, 1, 2, ... (document.anchors.length-1).) When a button is pressed in the first window, the onClick event handler verifies that the anchor exists before setting window2.location.hash to the specified anchor name.

link1.html, which defines the first window and its buttons, contains the following code:

```
<HTML>
<HEAD>
<TITLE>Links and Anchors: Window 1</TITLE>
</HEAD>
<BODY>
<SCRIPT>
window2=open("link2.html","secondLinkWindow",
  "scrollbars=yes,width=250, height=400")
function linkToWindow(num) {
  if (window2.document.anchors.length > num)
    window2.location.hash=num
  else
    alert("Anchor does not exist!")
}
</SCRIPT>
<B>Links and Anchors</B>
<FORM>
<P>Click a button to display that anchor in window #2
<P><INPUT TYPE="button" VALUE="0" NAME="link0_button"
  onClick="linkToWindow(this.value)">
<INPUT TYPE="button" VALUE="1" NAME="link0_button"
  onClick="linkToWindow(this.value)">
<INPUT TYPE="button" VALUE="2" NAME="link0_button"
  onClick="linkToWindow(this.value)">
<INPUT TYPE="button" VALUE="3" NAME="link0_button"
  onClick="linkToWindow(this.value)">
<INPUT TYPE="button" VALUE="4" NAME="link0_button"
  onClick="linkToWindow(this.value)">
</FORM>
</BODY>
</HTML>
```

link2.html, which contains the anchors, contains the following code:

```
<HTML>
<HEAD>
<TITLE>Links and Anchors: Window 2</TITLE>
</HEAD>
```

```
<BODY>
<A NAME="0"><B>Some numbers</B> (Anchor 0)</A>
<UL><LI>one
<LI>two
<LI>three
<LI>four</UL>
<P><A NAME="1"><B>Some colors</B> (Anchor 1)</A>
<UL><LI>red
<LI>orange
<LI>yellow
<LI>green</UL>
<P><A NAME="2"><B>Some music types</B> (Anchor 2)</A>
<UL><LI>R&B
<LI>Jazz
<LI>Soul
<LI>Reggae
<LI>Rock</UL>
<P><A NAME="3"><B>Some countries</B> (Anchor 3)</A>
<UL><LI>Afghanistan
<LI>Brazil
<LI>Canada
<LI>Finland
<LI>India</UL>
</BODY>
</HTML>
```

- ## Applet

Includes a Java applet in a web page.

Created by

The HTML APPLET tag. The JavaScript runtime engine creates an Applet object corresponding to each applet in your document. It puts these objects in an array in the document.applets property. You access an Applet object by indexing this array.

To define an applet, use standard HTML syntax. If you specify the NAME attribute, you can use the value of that attribute to index into the applets array. To refer to an applet in JavaScript, you must supply the MAYSCRIPT attribute in its definition.

Description

The author of an HTML page must permit an applet to access JavaScript by specifying the MAYSCRIPT attribute of the APPLET tag. This prevents an applet from accessing JavaScript on a page without the knowledge of the page author. For example, to allow the musicPicker.class applet access to JavaScript on your page, specify the following:

```
<APPLET CODE="musicPicker.class" WIDTH=200 HEIGHT=35
  NAME="musicApp" MAYSCRIPT>
```

Accessing JavaScript when the MAYSCRIPT attribute is not specified results in an exception.

Property Summary
All public properties of the applet are available for JavaScript access to the Applet object.

Method Summary
All public methods of the applet

Examples
The following code launches an applet called musicApp:

```
<APPLET CODE="musicSelect.class" WIDTH=200 HEIGHT=35
  NAME="musicApp" MAYSCRIPT>
</APPLET>
```

- **Area**

Defines an area of an image as an image map. When the user clicks the area, the area's hypertext reference is loaded into its target window. Area objects are a type of Link object.

- **document**

Contains information about the current document, and provides methods for displaying HTML output to the user.

The onBlur, onFocus, onLoad, and onUnload event handlers are specified in the BODY tag but are actually event handlers for the Window object. The following are event handlers for the document object.

- onClick
- onDblClick
- onKeyDown
- onKeyPress
- onKeyUp
- onMouseDown
- onMouseUp

document.forms	Returns a collection of all <form> elements in the document
document.getElementById()	Returns the element that has the ID attribute with the specified value

document.getElementsByClassName()	Returns a NodeList containing all elements with the specified class name
document.getElementsByName()	Returns a NodeList containing all elements with a specified name
document.getElementsByTagName()	Returns a NodeList containing all elements with the specified tag name
document.hasFocus()	Returns a Boolean value indicating whether the document has focus

Other Property

alinkColor	A string that specifies the ALINK attribute.
anchors	An array containing an entry for each anchor in the document.
applets	An array containing an entry for each applet in the document.
bgColor	A string that specifies the BGCOLOR attribute.
cookie	Specifies a cookie.
domain	Specifies the domain name of the server that served a document.
embeds	An array containing an entry for each plug-in in the document.
fgColor	A string that specifies the TEXT attribute.
formName	A separate property for each named form in the document.
forms	An array a containing an entry for each form in the document.
images	An array containing an entry for each image in the document.

lastModified	A string that specifies the date the document was last modified.
layers	Array containing an entry for each layer within the document.
linkColor	A string that specifies the LINK attribute.
links	An array containing an entry for each link in the document.
plugins	An array containing an entry for each plug-in in the document.
referrer	A string that specifies the URL of the calling document.
title	A string that specifies the contents of the TITLE tag.
URL	A string that specifies the complete URL of a document.
vlinkColor	A string that specifies the VLINK attribute.

Method

captureEvents	Sets the document to capture all events of the specified type.
close	Closes an output stream and forces data to display.
getSelection	Returns a string containing the text of the current selection.
handleEvent	Invokes the handler for the specified event.
open	Opens a stream to collect the output of write or writeln methods.
releaseEvents	Sets the window or document to release captured events of the specified type, sending the event to objects further along the event hierarchy.
routeEvent	Passes a captured event along the normal event hierarchy.
write	Writes one or more HTML expressions to a document in the specified window.
writeln	Writes one or more HTML expressions to a document in the specified window and follows them with a newline character.

• Image

An image on an HTML form.

Client-side object

Created by
The Image constructor or the IMG tag.

The JavaScript runtime engine creates an Image object corresponding to each IMG tag in your document. It puts these objects in an array in the document.images property. You access an Image object by indexing this array.

To define an image with the IMG tag, use standard HTML syntax with the addition of JavaScript event handlers. If specify a value for the NAME attribute, you can use that name when indexing the images array.

To define an image with its constructor, use the following syntax:

new Image(width, height)

Parameters
width (Optional) The image width, in pixels.

height (Optional) The image height, in pixels.

Event handlers

- onAbort
- onError
- onKeyDown
- onKeyPress
- onKeyUp
- onLoad

To define an event handler for an Image object created with the Image constructor, set the appropriate property of the object. For example, if you have an Image object named imageName and you want to set one of its event handlers to a function whose name is handlerFunction, use one of the following statements:

imageName.onabort = handlerFunction
imageName.onerror = handlerFunction
imageName.onkeydown = handlerFunction
imageName.onkeypress = handlerFunction
imageName.onkeyup = handlerFunction
imageName.onload = handlerFunction
Image objects do not have onClick, onMouseOut, and onMouseOver event handlers. However, if you define an Area object for the image or place the IMG tag within a Link object, you can use the Area or Link object's event handlers. See Link.

Description
The position and size of an image in a document are set when the document is displayed in the web browser and cannot be changed using JavaScript (the width and height properties are read-

only for these objects). You can change which image is displayed by setting the src and lowsrc properties. (See the descriptions of Image.src and Image.lowsrc.)

You can use JavaScript to create an animation with an Image object by repeatedly setting the src property, as shown in Example 4 below. JavaScript animation is slower than GIF animation, because with GIF animation the entire animation is in one file; with JavaScript animation, each frame is in a separate file, and each file must be loaded across the network (host contacted and data transferred).

The primary use for an Image object created with the Image constructor is to load an image from the network (and decode it) before it is actually needed for display. Then when you need to display the image within an existing image cell, you can set the src property of the displayed image to the same value as that used for the previously fetched image, as follows.

```
myImage = new Image()
myImage.src = "seaotter.gif"
...
document.images[0].src = myImage.src
```

The resulting image will be obtained from cache, rather than loaded over the network, assuming that sufficient time has elapsed to load and decode the entire image. You can use this technique to create smooth animations, or you could display one of several images based on form input.

Property Summary

border	Reflects the BORDER attribute.
complete	Boolean value indicating whether the web browser has completed its attempt to load the image.
height	Reflects the HEIGHT attribute.
hspace	Reflects the HSPACE attribute.
lowsrc	Reflects the LOWSRC attribute.
name	Reflects the NAME attribute.
prototype	Allows the addition of properties to an Image object.
src	Reflects the SRC attribute.
vspace	Reflects the VSPACE attribute.
width	Reflects the WIDTH attribute.

Method Summary

handleEvent	Invokes the handler for the specified event.

Example 1: Create an image with theIMG **tag.** The following code defines an image using the IMG tag:

```
<IMG NAME="aircraft" SRC="f15e.gif" ALIGN="left" VSPACE="10">
```

The following code refers to the image:

```
document.aircraft.src='f15e.gif'
```

When you refer to an image by its name, you must include the form name if the image is on a form. The following code refers to the image if it is on a form:

```
document.myForm.aircraft.src='f15e.gif'
```

- ## Event handlers

Events are actions that occur usually as a result of something the user does. For example, clicking a button is an event, as is changing a text field or moving the mouse over a link. For your script to react to an event, you define *event handlers,* such as onChangeandonClick.

To create an event handler for an HTML tag, add an event handler attribute to the tag. Put JavaScript code in quotation marks as the attribute value. The general syntax is

```
<TAG eventHandler="JavaScript Code">
```

where TAG is an HTML tag and eventHandler is the name of the event handler. For example, suppose you have created a JavaScript function called compute. You can cause the browser to perform this function when the user clicks a button by assigning the function call to the button's onClick event handler:

```
<INPUT TYPE="button" VALUE="Calculate" onClick="compute(this.form)">
```

Table 9.2 Events and their corresponding event handlers.

Event	**Event handler**	**Event occurs when...**
abort	onAbort	The user aborts the loading of an image (for example by clicking a link or clicking the Stop button).
blur	onBlur	A form element loses focus or when a window or frame loses focus.
change	onChange	A select, text, or textarea field loses focus and its value has been modified.
click	onClick	An object on a form is clicked.
dblclick	onDblClick	The user double-clicks a form element or a link.
dragdrop	onDragDrop	The user drops an object onto the browser window, such as

		dropping a file on the browser window.
error	onError	The loading of a document or image causes an error.
focus	onFocus	A window, frame, or frameset receives focus or when a form element receives input focus.
keydown	onKeyDown	The user depresses a key.
keypress	onKeyPress	The user presses or holds down a key.
keyup	onKeyUp	The user releases a key.
load	onLoad	The browser finishes loading a window or all of the frames within a FRAMESETtag.
mousedown	onMouseDown	The user depresses a mouse button.
mousemove	onMouseMove	The user moves the cursor.
mouseout	onMouseOut	The cursor leaves an area (client-side image map) or link from inside that area or link.
mouseover	onMouseOver	The cursor moves over an object or area from outside that object or area.
mouseup	onMouseUp	The user releases a mouse button.
move	onMove	The user or script moves a window or frame.
reset	onReset	The user resets a form (clicks a Reset button).
resize	onResize	The user or script resizes a window or frame.
select	onSelect	The user selects some of the text within a text or textarea field.
submit	onSubmit	The user submits a form.
unload	onUnload	The user exits a document.

onAbort
Executes JavaScript code when an abort event occurs; that is, when the user aborts the loading of an image (for example by clicking a link or clicking the Stop button).

Syntax
onAbort="handlerText"

Examples

In the following example, an onAbort handler in an Image object displays a message when the user aborts the image load:

```
<IMG NAME="aircraft" SRC="f15e.gif"
   onAbort="alert('You didn\'t get to see the image!')">
```

onBlur

Executes JavaScript code when a blur event occurs; that is, when a form element loses focus or when a window or frame loses focus.

Syntax

onBlur="handlerText"

Parameters

Description

The blur event can result from a call to the Window.blur method or from the user clicking the mouse on another object or window or tabbing with the keyboard.

For windows, frames, and framesets, onBlur specifies JavaScript code to execute when a window loses focus.

Examples

Example 1: Validate form input. In the following example, userName is a required text field. When a user attempts to leave the field, the onBlur event handler calls the required function to confirm that userName has a legal value.

```
<INPUT TYPE="text" VALUE="" NAME="userName"
   onBlur="required(this.value)">
```

onChange

Executes JavaScript code when a change event occurs; that is, when a Select, Text, or Textarea field loses focus and its value has been modified.

Syntax

onChange="handlerText"

Description

Use onChange to validate data after it is modified by a user.

Examples

In the following example, userName is a text field. When a user changes the text and leaves the field, the onChange event handler calls the checkValue function to confirm that userName has a legal value.

```
<INPUT TYPE="text" VALUE="" NAME="userName"
   onChange="checkValue(this.value)">
```

onClick
Executes JavaScript code when a click event occurs; that is, when an object on a form is clicked. (A Click event is a combination of the MouseDown and MouseUp events).

Syntax
onClick="handlerText"

Description
For checkboxes, links, radio buttons, reset buttons, and submit buttons, onClick can return false to cancel the action normally associated with a click event.

Example 1: Call a function when a user clicks a button. Suppose you have created a JavaScript function called compute. You can execute the compute function when the user clicks a button by calling the function in the onClick event handler, as follows:

```
<INPUT TYPE="button" VALUE="Calculate" onClick="compute(this.form)">
```
In the preceding example, the keyword this refers to the current object; in this case, the Calculate button. The construct this.form refers to the form containing the button.

onDblClick
Executes JavaScript code when a DblClick event occurs; that is, when the user double-clicks a form element or a link.

Syntax
onDblClick="handlerText"

onDragDrop
Executes JavaScript code when a DragDrop event occurs; that is, when the user drops an object onto the browser window, such as dropping a file.

Event handler for Window

Implemented in Navigator 4.0

Syntax
onDragDrop="handlerText"

Description
The DragDrop event is fired whenever a system item (file, shortcut, and so on) is dropped onto the browser window using the native system's drag and drop mechanism. The normal response for the browser is to attempt to load the item into the browser window. If the event handler for the DragDrop event returns true, the browser loads the item normally. If the event handler returns false, the drag and drop is canceled.

onError
Executes JavaScript code when an error event occurs; that is, when the loading of a document or image causes an error.

Syntax
onError="handlerText"

Description
An error event occurs only when a JavaScript syntax or runtime error occurs, not when a browser error occurs. For example, if you try set window.location.href='notThere.html' and notThere.html does not exist, the resulting error message is a browser error message; therefore, onError would not intercept that message. However, an error event *is* triggered by a bad URL within an IMG tag or by corrupted image data.

window.onerror applies only to errors that occur in the window containing window.onerror, not in other windows.

onError can be any of the following:

- null to suppress all JavaScript error dialogs. Setting window.onerror to null means your users won't see JavaScript errors caused by your own code.
- The name of a function that handles errors (arguments are message text, URL, and line number of the offending line). To suppress the standard JavaScript error dialog, the function must return true. See Example 3 below.
- A variable or property that contains null or a valid function reference.

Example 1: Null event handler. In the following IMG tag, the code onError="null" suppresses error messages if errors occur when the image loads.

```
<IMG NAME="imageBad1" SRC="corrupt.gif" ALIGN="left" BORDER="2"
  onError="null">
```

onFocus
Executes JavaScript code when a focus event occurs; that is, when a window, frame, or frameset receives focus or when a form element receives input focus.

Syntax
onFocus="handlerText"

Description
The focus event can result from a focus method or from the user clicking the mouse on an object or window or tabbing with the keyboard. Selecting within a field results in a select event, not a focus event. onFocus executes JavaScript code when a focus event occurs.

A frame's onFocus event handler overrides an onFocus event handler in the BODY tag of the document loaded into frame.

Note that placing an alert in an onFocus event handler results in recurrent alerts: when you press OK to dismiss the alert, the underlying window gains focus again and produces another focus event.

Examples

The following example uses an onFocus handler in the valueFieldTextarea object to call the valueCheck function.

```
<INPUT TYPE="textarea" VALUE="" NAME="valueField"
   onFocus="valueCheck()">
```

onKeyDown

Syntax

onKeyDown="handlerText"

Parameters

handlerText JavaScript code or a call to a JavaScript function.

Description

A KeyDown event always occurs before a KeyPress event. If onKeyDown returns false, no KeyPress events occur. This prevents KeyPress events occurring due to the user holding down a key.

onKeyPress

Executes JavaScript code when a KeyPress event occurs; that is, when the user presses or holds down a key.

Syntax

onKeyPress="handlerText"

Parameters

Event properties used

Description

A KeyPress event occurs immediately after a KeyDown event only if onKeyDown returns something other than false. A KeyPress event repeatedly occurs until the user releases the key. You can cancel individual KeyPress events.

onKeyUp

Executes JavaScript code when a KeyUp event occurs; that is, when the user releases a key.

Syntax

onKeyUp="handlerText"

onLoad
Executes JavaScript code when a load event occurs; that is, when the browser finishes loading a window or all frames within a FRAMESET tag.

Syntax
onLoad="handlerText"

Description
Use the onLoad event handler within either the BODY or the FRAMESET tag, for example, <BODY onLoad="...">.

Example 1: Display message when page loads. In the following example, the onLoad event handler displays a greeting message after a Web page is loaded.

<BODY onLoad="window.alert("Welcome to the Brave New World home page!")>

onMouseDown
Executes JavaScript code when a MouseDown event occurs; that is, when the user depresses a mouse button.

Syntax
onMouseDown="handlerText"

Description
If onMouseDown returns false, the default action (entering drag mode, entering selection mode, or arming a link) is canceled.

Arming is caused by a MouseDown over a link. When a link is armed it changes color to represent its new state.

onMouseMove
Executes JavaScript code when a MouseMove event occurs; that is, when the user moves the cursor.

Syntax
onMouseMove="handlerText"

Description

The MouseMove event is sent only when a capture of the event is requested by an object

onMouseOut
Executes JavaScript code when a MouseOut event occurs; that is, each time the mouse pointer leaves an area (client-side image map) or link from inside that area or link.

Syntax
onMouseOut="handlerText"

Description
If the mouse moves from one area into another in a client-side image map, you'll get onMouseOut for the first area, then onMouseOver for the second.

Area objects that use the onMouseOut event handler must include the HREF attribute within the AREA tag.

onMouseOver
Executes JavaScript code when a MouseOver event occurs; that is, once each time the mouse pointer moves over an object or area from outside that object or area.

Syntax
onMouseOver="handlerText"

Description
If the mouse moves from one area into another in a client-side image map, you'll get onMouseOut for the first area, then onMouseOver for the second.

By default, the HREF value of an anchor displays in the status bar at the bottom of the browser when a user places the mouse pointer over the anchor. In the following example, onMouseOver provides the custom message "Click this if you dare."

```
<A HREF="http://home.netscape.com/"
   onMouseOver="window.status='Click this if you dare!'; return true">
Click me</A>
```

onMouseUp
Executes JavaScript code when a MouseUp event occurs; that is, when the user releases a mouse button.

Syntax
onMouseUp="handlerText"

Description
If onMouseUp returns false, the default action is canceled. For example, if onMouseUp returns false over an armed link, the link is not triggered. Also, if MouseUp occurs over an unarmed link (possibly due to onMouseDown returning false), the link is not triggered.

onMove
Executes JavaScript code when a move event occurs; that is, when the user or script moves a window or frame.

Syntax

onMove="handlerText"

onReset
Executes JavaScript code when a reset event occurs; that is, when a user resets a form (clicks a Reset button). Syntax

```
onReset="handlerText"
```

The following example displays a Text object with the default value "CA" and a reset button. If the user types a state abbreviation in the Text object and then clicks the reset button, the original value of "CA" is restored. The form's onReset event handler displays a message indicating that defaults have been restored.

```
<FORM NAME="form1" onReset="alert('Defaults have been restored.')">
State:
<INPUT TYPE="text" NAME="state" VALUE="CA" SIZE="2"><P>
<INPUT TYPE="reset" VALUE="Clear Form" NAME="reset1">
</FORM>
```

onResize
Executes JavaScript code when a resize event occurs; that is, when a user or script resizes a window or frame.

Syntax

```
onResize="handlerText"
```

Description

This event is sent after HTML layout completes within the new window inner dimensions. This allows positioned elements and named anchors to have their final sizes and locations queried, image SRC properties can be restored dynamically, and so on.

onSelect
Executes JavaScript code when a select event occurs; that is, when a user selects some of the text within a text or textarea field. Syntax

```
onSelect="handlerText"
```

The following example uses onSelect in the valueFieldText object to call the selectState function.

```
<INPUT TYPE="text" VALUE="" NAME="valueField" onSelect="selectState()">
```

onSubmit
Executes JavaScript code when a submit event occurs; that is, when a user submits a form.

Syntax
onSubmit="handlerText"

Parameters

Description
You can use onSubmit to prevent a form from being submitted; to do so, put a return statement that returns false in the event handler. Any other returned value lets the form submit. If you omit the return statement, the form is submitted.

In the following example, onSubmit calls the validate function to evaluate the data being submitted. If the data is valid, the form is submitted; otherwise, the form is not submitted.

```
<FORM onSubmit="return validate(this)">
...
</FORM>
```

onUnload
Executes JavaScript code when an unload event occurs; that is, when the user exits a document.

Syntax
onUnload="handlerText"

Description
Use onUnload within either the BODY or the FRAMESET tag, for example, <BODY onUnload="...">.

In the following example, onUnload calls the cleanUp function to perform some shutdown processing when the user exits a Web page:

```
<BODY onUnload="cleanUp()">
```