JavaScript in one page

Review

JavaScript is a compact, object-based scripting language for developing client and server Internet applications. Netscape Navigator interprets JavaScript statements embedded in an HTML page, and LiveWire enables you to create server-based applications similar to Common Gateway Interface (CGI) programs.

JavaScript is Netscape's cross-platform, object-based scripting language for client and server applications. There are two types of JavaScript:

- Navigator JavaScript, also called client-side JavaScript
- LiveWire JavaScript, also called server-side JavaScript

JavaScript is a language. Client and server JavaScript differ in numerous ways, but they have the following elements in common:

- Keywords, statement syntax, and grammar
- Rules for expressions, variables, and literals
- Underlying object model (although Navigator and LiveWire have different object frameworks)
- Built-in objects and functions

Allocation

```
<html lang="en">

 <head>
  <title>... replace with your document's title ...</title>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

  <script type="text/javascript" src="... insert link to file with your JavaScript code here ..."></script>

  <script type="text/javascript">
   //<!-- Begin to hide script contents from old browsers.
   ... or insert your JavaScript code here ...
   // End the hiding here. -->
  </script>

 </head>

 <body>
  ... replace with your document's content ...

  <script type="text/javascript">
   //<!-- Begin to hide script contents from old browsers.
   ... or insert your JavaScript code here ...
   // End the hiding here. -->
  </script>

  ... replace with your document's content ...

  <input type="button" value="Click Me" onClick="... or insert your JavaScript code here ..." />

  <a href="... any link or sharp ..." onBlur="... or insert your JavaScript code here ...">... replace with your text ...</a>

  ... replace with your document's content ...
 </body>

</html>
```

JavaScript Simple Examples

Hello World!

Code:

```
<html>
  <head></head>
  <body>
    <strong>Example:</strong>
    <script type="text/javascript">
      //<!--
      document.write("Hello World!");
      //-->
    </script>
    <div>All done.</div>
  </body>
</html>
```

Example:

Hello World!
All done.

Defining and Dalling Functions

Code:

```
<html>
  <head>
    <script type="text/javascript">
      //<!--
      function square(number) {
        return number * number;
      }
      //-->
    </script>
  </head>
  <body>
    <strong>Example:</strong>
    <script type="text/javascript">
      //<!--
      document.write("The function");
      document.write(" returned ");
      document.write(square(5), ".");
      //-->
    </script>
    <div>All done.</div>

  </body>
</html>
```

Example:

The function returned 25.
All done.

Using an Event Handler

Code:

```
<html>
  <head>
    <script type="text/javascript">
      //<!--
      function compute(f) {
        if (confirm("Are you sure?"))
          f.result.value = eval(f.expr.value)
        else
          alert("Please come back again.")
      }
      //-->
    </script>
  </head>
  <body>
    <strong>Example:</strong>
    <form>
      Enter an expression:
      <input type="text" name="expr" size="10"
value="2+2" /><br/>
      <input type="button" value="Calculate"
onClick="compute(this.form)" /><br/>
      Result:
      <input type="text" name="result" size="10" /><br/>
    </form>
  </body>
</html>
```

Example:

Enter an expression: 2+2
Calculate
Result:

Values

JavaScript recognizes the following types of values:

- **Numbers**, such as *42* or *3.14159*
- **Logical (Boolean) values**, either *true* or *false*
- **Strings**, such as *"Howdy!"*
- **null**, a special keyword denoting a null value

Data type conversion

JavaScript is a loosely typed language. That means you do not have to specify the data type of a variable when you declare it, and data types are converted automatically as needed during script execution. So, for example, you could define a variable as follows:

```
var answer = 42
```

And later, you could assign the same variable a string value, for example,

```
answer = "Thanks for all the fish..."
```

Because JavaScript is loosely typed, this assignment does not cause an error message.

In expressions involving numeric and string values, JavaScript converts the numeric values to strings. For example, consider the following statements:

```
x = "The answer is " + 42
y = 42 + " is the answer."
```

The first statement returns the string "The answer is 42." The second statement returns the string "42 is the answer."

Variables

You use variables as symbolic names for values in your application. You give variables names by which you refer to them and which must conform to certain rules.

A JavaScript identifier, or name, must start with a letter or underscore ("_"); subsequent characters can also be digits (0-9). Because JavaScript is case sensitive, letters include the characters "A" through "Z" (uppercase) and the characters "a" through "z" (lowercase).

Some examples of legal names are Number_hits, temp99, and _name.

You can declare a variable in two ways:

- By simply assigning it a value; for example, *x = 42*
- With the keyword var; for example, *var x = 42*

When you set a variable identifier by assignment outside of a function, it is called a global variable, because it is available everywhere in the current document. When you declare a variable within a function, it is called a local variable, because it is available only within the function. Using var is optional, but you need to use it if you want to declare a local variable inside a function that has already been declared as a global variable.

You can access global variables declared in one window or frame from another window or frame by specifying the window or frame name. For example, if a variable called phoneNumber is declared in a FRAMESET document, you can refer to this variable from a child frame as parent.phoneNumber.

Literals

You use literals to represent values in JavaScript. These are fixed values, not variables, that you literally provide in your script.

Integers

Integers can be expressed in decimal (base 10), hexadecimal (base 16), and octal (base 8). A decimal integer literal consists of a sequence of digits without a leading 0 (zero). A leading 0 (zero) on an integer literal indicates it is in octal; a leading 0x (or 0X) indicates hexadecimal. Hexadecimal integers can include digits (0-9) and the letters a-f and A-F. Octal integers can include only the digits 0-7. Some examples of integer literals are: *42*, *0xFFF*, and *-345*.

Floating-point literals

A floating-point literal can have the following parts: a decimal integer, a decimal point ("."), a fraction (another decimal number), an exponent, and a type suffix. The exponent part is an "e" or "E" followed by an integer, which can be signed (preceded by "+" or "-"). A floating-point literal must have at least one digit, plus either a decimal point or "e" (or "E"). Some examples of floating-point literals are *3.1415*, *-3.1E12*, *.1e12*, and *2E-12*

Boolean literals

The Boolean type has two literal values: *true* and *false*.

String literals

A string literal is zero or more characters enclosed in double (") or single (') quotation marks. A string must be delimited by quotation marks of the same type; that is, either both single quotation marks or double quotation marks.

The following are examples of string literals: *"blah"*, *'blah'*, *"1234"*, *"one line \n another line"*.

In addition to ordinary characters, you can also include special characters in strings, as shown in the last element in the preceding list.

Special Characters

Character	Meaning
\b	backspace
\f	form feed
\n	new line
\r	carriage return
\t	tab
\\	backslash character

For characters not listed in the preceding table, a preceding backslash is ignored, with the exception of a quotation mark and the backslash character itself. You can insert quotation marks inside strings by preceding them with a backslash. This is known as escaping the quotation marks. For example,

```
var quote = "He read \"The Cremation of Sam McGee\" by R.W. Service."
document.write(quote)
```

The result of this would be He read "The Cremation of Sam McGee" by R.W. Service. To include a literal backslash inside a string, you must escape the backslash character. For example, to assign the file path c:\temp to a string, use the following:

```
var home = "c:\\temp"
```

Expressions

An expression is any valid set of literals, variables, operators, and expressions that evaluates to a single value; the value can be a number, a string, or a logical value.

Conceptually, there are two types of expressions: those that assign a value to a variable, and those that simply have a value.

For example, the expression *x = 7* is an expression that assigns x the value seven.

This expression itself evaluates to seven. Such expressions use assignment operators. On the other hand, the expression *3 + 4* simply evaluates to seven; it does not perform an assignment.

The operators used in such expressions are referred to simply as operators.

JavaScript has the following types of expressions:

- **Arithmetic:** evaluates to a number, for example *3.14159*
- **String:** evaluates to a character string, for example, *"Fred"* or *"234"*
- **Logical:** evaluates to *true* or *false*

The special keyword null denotes a null value. In contrast, variables that have not been assigned a value are undefined and will cause a runtime error if used as numbers or as numeric variables.
Array elements that have not been assigned a value, however, evaluate to false. For example, the following code executes the function myFunction because the array element is not defined:

```
myArray=new Array()
if (!myArray["notThere"])
myFunction()
```

A **conditional expression** can have one of two values based on a condition. The syntax is

```
(condition) ? val1 : val2
```

If condition is true, the expression has the value of val1. Otherwise it has the value of val2. You can use a conditional expression anywhere you would use a standard expression.
For example,

```
status = (age >= 18) ? "adult" : "minor"
```

This statement assigns the value "adult" to the variable status if age is eighteen or greater. Otherwise, it assigns the value "minor" to status.

Operators

JavaScript has assignment, comparison, arithmetic, bitwise, logical, string, and special operators. This section describes the operators and contains information about operator precedence.
There are both binary and unary operators. A binary operator requires two operands, one before the operator and one after the operator:

```
operand1 operator operand2
```

For example, 3+4 or *x*y*.
A unary operator requires a single operand, either before or after the operator:

```
operator operand
```

or

```
operand operator
```

For example, x++ or ++x.

Assignment operators

An assignment operator assigns a value to its left operand based on the value of its right operand.
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 operators are shorthand for standard operations, as shown in the following table:

Shorthand 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

Comparison operators

A comparison operator compares its operands and returns a logical value based on whether the comparison is true or not.
The operands can be numerical or string values. When used on string values, the comparisons are based on the standard lexicographical ordering.
They are described in the following table.

Comparison operators

Operator	Name	Description	Example	Example Descritpion
==	Equal	Returns true if the operands are equal	*x == y*	Returns true if x equals y.
!=	Not equal	Returns true if the operands are not equal.	*x != y*	Returns true if x is not equal to y.
>	Greater than	Returns true if left operand is greater than right operand.	*x > y*	Returns true if x is greater than y.
>=	Greater than or equal	Returns true if left operand is greater than or equal to right operand.	*x >= y*	Returns true if x is greater than or equal to y.
<	Less than	Returns true if left operand is less than right operand.	*x < y*	Returns true if x is less than y.
<=	Less than or equal	Returns true if left operand is less than or equal to right operand.	*x <= y*	Returns true if x is less than or equal to y.

Logical operators

Logical operators take Boolean (logical) values as operands and return a Boolean value. They are described in the following table.

Logical Operators

Operator	Name	Usage	Description
&&	and	expr1 && expr2	Returns true if both logical expressions expr1 and expr2 are true. Otherwise, returns false.
||	or	expr1 || expr2	Returns true if either logical expression expr1 or expr2 is true. If both are false, returns false.
!	not	!expr	If expr is true, returns false; if expr is false, returns true.

Notes

- **false** && anything is short-circuit evaluated to false.
- **true** || anything is short-circuit evaluated to true.

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 (/).
These operators work as they do in other programming languages.

Arithmetic operators

Synopsis	Name	Description	Example	Example Description
var1 % var2	Modulus	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 floating-point remainder of dividing var1 by var2	*13 % 5*	Returns 3
var++	Increment	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.	*y = x++*	If x is three, then the statement y = x++ sets y to three and increments x to four.
++var			*y=++x*	If x is three, then the statement y = ++x increments x to four and sets y to four.
var--	Decrement	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.	*y = x--*	If x is three, then the statement y = x-- sets y to three and decrements x to two.
--var			*y=--x*	If x is three, then the statement y = --x decrements x to two and sets y to two.
-var	Unary negation	The unary negation precedes its operand and negates it.	*x = -x*	Negates the value of x; that is, if x were three, it would become -3.

Bitwise operators

Bitwise operators treat their operands as a set of 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.

Bitwise operators

Operator	Usage	Description	Example
Bitwise AND	*a & b*	Returns a one in each bit position if bits of both operands are ones.	15 & 9 yields 9 (1111 & 1001 = 1001)
Bitwise OR	*a | b*	Returns a one in a bit if bits of either operand is one.	15 | 9 yields 15 (1111 | 1001 = 1111)
Bitwise XOR	*a ^ b*	Returns a one in a bit position if bits of one but not both operands are one.	15 ^ 9 yields 6 (1111 ^ 1001 = 0110)
Bitwise NOT	*~ a*	Flips the bits of its operand.	
Left shift	*a << b*	Shifts a in binary representation b bits to left, shifting in zeros from the right.	9<<2 yields thirty-six, because 1001 shifted two bits to the left becomes 100100, which is thirty-six.
Sign-propagating right shift	*a >> b*	Shifts a in binary representation b bits to right, discarding bits shifted off.	9>>2 yields two, because 1001 shifted two bits to the right becomes 10, which is two.
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.	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.

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".
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

Name	Synopsis	Description	Example
new	*objectName = new objectType (param1 [,param2] ...[,paramN])*	You can use the new operator to create an instance of a user-defined object type or of one of the built-in object types Array, Boolean, Date, Function, Math, Number, or String.	
typeof	*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
	typeof (operand)		
void	*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.	 Click here to do nothing
	javascript:void expression		 Click here to submit

Operator precedence

The precedence of operators determines the order they are applied when evaluating an expression. You can override operator precedence by using parentheses.
The following table describes the precedence of operators, from lowest to highest:

Operator precedence

Operator	Examples
assignment	= += -= *= /= %= <<= >>= >>>= &= ^= |=
conditional	?:
logical-or	||
logical-and	&&
bitwise-or	|
bitwise-xor	^
bitwise-and	&
equality	== !=
relational	< <= > >=
bitwise shift	<< >> >>>
addition/subtraction	+ -
multiply/divide	* / %
negation/increment	! ~ - ++ -- typeof void
call, member	() [] .

Statements (in alphabetical order)

break

break
A statement that terminates the current while or for loop and transfers program control to the statement following the terminated loop.

```
/* The following function has a break statement that terminates the while loop when i 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;
}
```

1. // comment text
2. /* multiple line comment text */

comment

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

```
// 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. */
```

continue

continue

A statement that terminates execution of the block of statements in a while or for loop, and continues execution of the loop with the next iteration. 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 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;
}
```

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

for

A statement that 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.

Arguments

- *initial-expression* is a statement or variable declaration. It is typically used to initialize a counter variable. This expression may optionally declare new variables with the var keyword.
- *condition* is 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* is generally used to update or increment the counter variable.
- *statements* is a 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.

```
    /* The following for statement starts by declaring the variable i and initializing it to zero. It checks that i is less than
nine, performs the two succeeding statements, and increments i by one after each pass through the loop. */

for (var i = 0; i < 9; i++) {
 n += i;
 myfunc(n);
}
```

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

for...in

A statement that iterates a specified variable over all the properties of an object. For each distinct property, JavaScript executes the specified statements.

Arguments

- *variable* is the variable to iterate over every property.
- *object* is the object for which the properties are iterated.
- *statements* specifies the statements to execute for each property.

```
    /* 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, obj_name) {
 var result = "";
 for (var i in obj) {
  result += obj_name + "." + i + " = " + obj[i] + "<br />";
 }
 result += "<hr>";
 return result;
}
```

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

function

A statement that declares a JavaScript function *name* with the specified parameters param. Acceptable parameters include strings, numbers, and objects.

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.

Arguments

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

```
/* 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;
}
```

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

if...else

A statement that executes a set of statements if a specified condition is true. If the condition is false, another set of statements can be executed.

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* and *statements2* can be any JavaScript statements, including further nested **if** statements. Multiple statements must be enclosed in braces.

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

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

new

An operator that lets you create an instance of a user-defined object type or of one of the built-in object types *Array*, *Boolean*, *Date*, *Function*, *Math*, *Number*, or *String*.

Creating a user-defined object type requires two steps:

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.

Arguments

- *objectName* is the name of the new object instance.
- *objectType* is the object type. It must be a function that defines an object type.
- *param1*...*paramN* are the property values for the object. These properties are parameters defined for the *objectType* function.

```
    /* 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;
}
mycar = new car("Eagle", "Talon TSi", 1993);

    /* 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
```

return *expression*

return

A statement that specifies the value to be returned by a function.

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

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

switch(*variable*) {
case *value_1*:
***statements_1*;**
break;
case *value_1*:
***statements_2*;**
break;
...
default:
***statements_default*;**
}

switch

The switch statement can be used for multiple branches based on a number or string.

Arguments:

- *variable* is any variable.
- *value_...* is any valid value.
- *statements_...* is any block of statements.

```
switch(a) {
 case 1:    doit();
  break;
 case 2:
  doit2();
  break;
 default:
  donothing();
}
```

this[.*propertyName*]

this

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

```
    /* 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!")
}
```

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

var

A statement that declares a variable, optionally initializing it to a value. 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.

Arguments

- *varname* is the variable name. It can be any legal identifier.
- *value* is the initial value of the variable and can be any legal expression.

```
/* 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)" />
var num_hits = 0, cust_no = 0
```

while (*condition*) {
** *statements***
}

do {
** *statements***
} while (*condition*)

while

A statement that 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.

Arguments

- *condition* is 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* is a 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 **while** statement.

```
/* 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 (*object*) {
** *statements***
}

with

A statement that 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.

Arguments

- *object* specifies the default *object* to use for the *statements*. The parentheses around *object* are required.
- *statements* is any block of statements.

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);
}
```

Functions

Functions are one of the fundamental building blocks in JavaScript. A function is a JavaScript procedure - a set of statements that performs a specific task. To use a function, you must first define it; then your script can call it.

Defining functions

A function definition consists of the function keyword, followed by

- The name of the function.
- A list of arguments to the function, enclosed in parentheses and separated by commas.
- The JavaScript statements that define the function, enclosed in curly braces, { }. The statements in a function can include calls to other functions defined in the current application.

In Navigator JavaScript, it is good practice to define all your functions in the HEAD of a page so that when a user loads the page, the functions are loaded first.

For example, here is the definition of a simple function named pretty_print:

```
function pretty_print(str) {
  document.write("<hr><p>" + str)
}
```

, or semantically equivalent:

```
var pretty_print = function(str) {
  document.write("<hr><p>" + str)
}
```

This function takes a string, str, as its argument, adds some HTML tags to it using the concatenation operator (+), and then displays the result to the current document using the write method.

Using functions

In a Navigator application, you can use (or call) any function defined in the current page. You can also use functions defined by other named windows or frames. In a LiveWire application, you can use any function compiled with the application.

Defining a function does not execute it. You have to call the function for it to do its work. For example, if you defined the example function pretty_print in the HEAD of the document, you could call it as follows:

```
<script type="text/javascript">
  pretty_print("This is some text to display")
</script>
```

The arguments of a function are not limited to strings and numbers. You can pass whole objects to a function, too.

A function can even be recursive, that is, it can call itself. For example, here is a function that computes factorials:

```
function factorial(n) {
  if ((n == 0) || (n == 1)) return 1
  else {
    result = (n * factorial(n-1) )
    return result
  }
}
```

You could then display the factorials of one through five as follows:

```
for (x = 0; x < 5; x++) {
  document.write("<br />", x, " factorial is ", factorial(x))
}
```

The results are:

```
0 factorial is 1
1 factorial is 1
2 factorial is 2
3 factorial is 6
4 factorial is 24
5 factorial is 120
```

Using the arguments array

The arguments of a function are maintained in an array. Within a function, you can address the parameters passed to it as follows:

```
functionName.arguments[i]
```

where functionName is the name of the function and i is the ordinal number of the argument, starting at zero. So, the first argument passed to a function named myfunc would be myfunc.arguments[0]. The total number of arguments is indicated by the variable arguments.length.

Using the arguments array, you can call a function with more arguments than it is formally declared to accept using. This is often useful if you don't know in advance how many arguments will be passed to the function. You can use arguments.length to determine the number of arguments actually passed to the function, and then treat each argument using the arguments array.

For example, consider a function defined to create HTML lists. The only formal argument for the function is a string that is "U" for an unordered (bulleted) list or "O" for an ordered (numbered) list. The function is defined as follows:

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

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

```
list("o", "one", 1967, "three", "etc., etc...")
```

results in this output:

```
1.one
2.1967
3.three
4.etc., etc...
```

Built-in Functions

isNaN(*testValue*)

isNaN function

The **isNaN** function evaluates an argument to determine if it is "NaN" (not a number).

Arguments

- *testValue* is the value you want to evaluate.

On platforms that support NaN, 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.

```
/* The following code evaluates floatValue to determine if it is a number and then calls a procedure accordingly: */
floatValue=parseFloat(toFloat);
if (isNaN(floatValue)) {
  notFloat();
} else {
  isFloat();
}
```

parseFloat(*str*)

parseFloat

parseFloat parses its argument, the string str, and attempts to return a floating-point number. If it encounters a character other than a sign (+ or -), a numeral (0-9), a decimal point, or an exponent, then it returns the value up to that point and ignores that character and all succeeding characters. If the first character cannot be converted to a number, it returns "NaN" (not a number).

```
parseFloat("5.347")
```

parseInt(*str* [, *radix*])

parseInt

parseInt parses its first argument, the string str, and attempts to return an integer of the specified radix (base), indicated by the second, optional argument, radix. For example, a radix of ten indicates to convert to a decimal number, eight octal, sixteen hexadecimal, and so on. For radixes above ten, the letters of the alphabet indicate numerals greater than nine. 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. If the first character cannot be converted to a number in the specified radix, it returns "NaN." The **parseInt** function truncates numbers to integer values.

```
parseInt("7")
```

Objects

JavaScript is based on a simple object-oriented paradigm.
An object is a construct with properties that are JavaScript variables or other objects.
An object also has functions associated with it that are known as the object's methods.
In addition to objects that are built into the Navigator client and the LiveWire server, you can define your own objects.

Creating new objects

Both client and server JavaScript have a number of predefined objects. In addition, you can create your own objects. Creating your own object requires two steps:

1. Define the object type by writing a constructor 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. For example, 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, year, and color. To do this, you would write the following function:

```
function car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}
```

Notice the use of this to assign values to the object's properties based on the values passed to the function.
Now you can create an object called mycar as follows:

Defining methods

A method is a function associated with an object. You define a method the same way you define a standard function. Then you use the following syntax to associate the function with an existing object:

```
object.methodname = function_name
```

where object is an existing object, methodname is the name you are assigning to the method, and function_name is the name of the function.
You can then call the method in the context of the object as follows:

```
object.methodname(params);
```

You can define methods for an object type by including a method definition in the object constructor function. For example, you could define a function that would format and display the properties of the previously-defined car objects; for example,

```
function displayCar() {
  var result = "A Beautiful " + this.year + " " + this.make + " " + this.model;
  pretty_print(result);
}
```

where pretty_print is the function (defined in "Functions") to display a horizontal rule and a string. Notice the use of this to refer to the object to which the method

```
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)
```

An object can have a property that is itself another object. For example, 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 McKinnon", 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)
```

Notice that 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 arguments for the owners. Then if you want to find out the name of the owner of car2, you can access the following property:

```
car2.owner.name
```

Note that 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 have to add the property to the definition of the car object type.
belongs.
You can make this function a method of car by adding the statement

```
this.displayCar = displayCar;
```

to the object definition. So, the full definition of car would now look like

```
function car(make, model, year, owner) {   this.make = make;
  this.model = model;
  this.year = year;
  this.owner = owner;
  this.displayCar = displayCar;
}
```

Then you can call the displayCar method for each of the objects as follows:

```
car1.displayCar()
car2.displayCar()
```

This will produce output like:

```
A Beautiful 1993 Eagle Talon TSi
A Beautiful 1992 Nissan 300ZX
```

Defining object with "return"

Let's consider a person object with first and last name fields. There are two ways in which their name might be displayed: as "first last" or as "last, first".

```
function Person(first, last) {
  return {
    first: first,
    last: last,
    fullName: function() {
      return this.first + ' ' + this.last;
    },
    fullNameReversed: function() {
      return this.last + ', ' + this.first;
    }
  }
}
```

use (with trace):

```
> s = Person("Simon", "Willison")
> s.fullName()
Simon Willison
> s.fullNameReversed()
Willison, Simon
```

Defining object with "this"

```
function Person(first, last) {
  this.first = first;
  this.last = last;
  this.fullName = function() {
    return this.first + ' ' + this.last;
  }
  this.fullNameReversed = function() {
    return this.last + ', ' + this.first;
  }
}
```

, or:

```
function personFullName() {
    return this.first + ' ' + this.last;
  }
function personFullNameReversed() {
    return this.last + ', ' + this.first;
  }

function Person(first, last) {
  this.first = first;
  this.last = last;
  this.fullName = personFullName
  this.fullNameReversed = personFullNameReversed
}
```

use (with trace):

```
> s = new Person("Simon", "Willison")
> s.fullName()
Simon Willison
> s.fullNameReversed()
Willison, Simon
```

Defining object with "prototype"

```
function Person(first, last) {
  this.first = first;
  this.last = last;
}

Person.prototype.fullName = function() {
  return this.first + ' ' + this.last;
}
Person.prototype.fullNameReversed = function() {
  return this.last + ', ' + this.first;
}
```

use (with trace):

```
> s = new Person("Simon", "Willison")
> s.fullName()
Simon Willison
> s.fullNameReversed()
Willison, Simon
```

This is an incredibly powerful tool. JavaScript lets you modify something's prototype at any time in your program, which means you can add extra methods to existing objects at runtime:

```
> s = new Person("Simon", "Willison");
> s.firstNameCaps();
TypeError on line 1: s.firstNameCaps is not a function
> Person.prototype.firstNameCaps = function() {
>   return this.first.toUpperCase()
> }
> s.firstNameCaps()
SIMON
```

Built-in Objects

JavaScript Root Object Properties (for all built-in objects)

constructor A reference to the function that created the object

Example №1
In this example we will show how to use the constructor property:

```
var test=new Array();
if (test.constructor==Array) {document.write("This is an Array")};
if (test.constructor==Boolean) {document.write("This is a Boolean")};
if (test.constructor==Date) {document.write("This is a Date")};
if (test.constructor==String) {document.write("This is a String")}
```

The output of the code above will be:

```
This is an Array
```

Example 2
In this example we will show how to use the constructor property:

```
function employee(name,jobtitle,born) {this.name=name; this.jobtitle=jobtitle; this.born=born;}
var fred=new employee("Fred Flintstone","Caveman",1970);
document.write(fred.constructor);
```

The output of the code above will be:

```
function employee(name,jobtitle,born) { this.name=name; this.jobtitle=jobtitle ; this.born=born; }
```

prototype Lets you add a properties to an object.

Example
In this example we will show how to use the prototype property to add a property to an object:

```
function employee(name,jobtitle,born) {this.name=name; this.jobtitle=jobtitle; this.born=born;}
var fred=new employee("Fred Flintstone","Caveman",1970);
employee.prototype.salary=null;
fred.salary=20000;
document.write(fred.salary);
```

The output of the code above will be:

```
20000
```

JavaScript Array Object Description

Review
JavaScript does not have an explicit array data type. However, you can use the built-in Array object and its methods to work with arrays in your applications. The Array object has methods for joining, reversing, and sorting arrays. It has a property for determining the array length.
An array is an ordered set of values that you reference through a name and an index. For example, you could have an array called emp that contains employees' names indexed by their employee number. So emp[1] would be employee number one, emp[2] employee number two, and so on.
To create an Array object:

1. arrayObjectName = new Array([arrayLength])
2. arrayObjectName = new Array([element0, element1, ..., elementn])

length

concat (*arrayX,arrayX,.....,arrayX*)

JavaScript Root Object Methods (for all built-in objects)

eval(*string*)
Evaluates a string of JavaScript code in the context of the specified object.

- *string* is any string representing a JavaScript expression, statement, or sequence of statements. The expression can include variables and properties of existing objects.

toSource()
Represents the source code of an object

Note:

- This method does not work in Internet Explorer!

Example
In this example we will show how to use the toSource() method:

```
function employee(name,jobtitle,born) {
this.name=name; this.jobtitle=jobtitle; this.born=born;
}
var fred=new employee("Fred Flintstone","Caveman",1970);
document.write(fred.toSource());
```

The output of the code above will be:

toString()
Converts a Boolean value to a string and returns the result

Note:

- The elements in the object will be separated with commas.

Example
In this example we will create an array and convert it to a string:

```
var arr = new Array(3);
arr[0] = "Jani"; arr[1] = "Hege"; arr[2] = "Stale";
document.write(arr.toString());
```

The output of the code above will be:

```
Jani,Hege,Stale
```

valueOf()
Returns the primitive value of a object
The primitive value is inherited by all objects descended from the object.
The valueOf() method is usually called automatically by JavaScript behind the scenes and not explicitly in code.

JavaScript Array Object Properties

Reflects the number of elements in an array

JavaScript Array Object Methods

Joins two or more arrays and returns the result
This method does not change the existing arrays, it only returns a copy of the joined arrays.

Arguments

- *arrayX* - one or more array objects to be joined to an array

Example №1:
Here we create two arrays and show them as one using concat():

```
var arr = new Array(3)
arr[0] = "Jani"; arr[1] = "Tove";
arr[2] = "Hege";
var arr2 = new Array(3)
arr2[0] = "John";
arr2[1] = "Andy";
```

Example №2:
Here we create three arrays and show them as one using concat():

```
var arr = new Array(3)
arr[0]="Jani"; arr[1]="Tove"; arr[2]="Hege";
var arr2 = new Array(3)
arr2[0]="John"; arr2[1]="Andy"; arr2[2]="Wendy";
var arr3 = new Array(2)
arr3[0]="Stale"; arr3[1]="Borge";
```

Arguments

- *arrayObjectName*is either the name of a new object or a property of an existing object. When using Array properties and methods, arrayObjectName is either the name of an existing Array object or a property of an existing object.
- *arrayLength*this the initial length of the array. You can access this value using the length property.
- *elementn*is a list of values for the array's elements. When this form is specified, the array is initalized with the specified values as its elements, and the array's length property is set to the number of arguments.

The Array object has the following main methods:

- **join** - joins all elements of an array into a string
- **reverse** - transposes the elements of an array: the first array element becomes the last and the last becomes the first
- **sort** - sorts the elements of an array

For example, suppose you define the following array:

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

myArray.join() returns "Wind,Rain,Fire"; myArray.reverse transposes the array so that myArray[0] is "Fire", myArray[1] is "Rain", and myArray[2] is "Wind". myArray.sort sorts the array so that myArray[0] is "Fire", myArray[1] is "Rain", and myArray[2] is "Wind". myArray.

Defining Arrays

The Array object is used to store a set of values in a single variable name.

We define an Array object with the new keyword. The following code line defines an Array object called myArray:

```
var myArray=new Array()
```

There are two ways of adding values to an array (you can add as many values as you need to define as many variables you require).

1:

```
var mycars=new Array();
mycars[0]="Saab";
mycars[1]="Volvo";
mycars[2]="BMW"
```

You could also pass an integer argument to control the array's size:

```
var mycars=new Array(3);
mycars[0]="Saab";
mycars[1]="Volvo";
mycars[2]="BMW"
```

2:

```
var mycars=new Array("Saab","Volvo","BMW")
```

Note:

If you specify numbers or true/false values inside the array then the type of variables will be numeric or Boolean instead of string.

Accessing Arrays

You can refer to a particular element in an array by referring to the name of the array and the index number. The index number starts at 0.

The following code line:

```
document.write(mycars[0])
```

will result in the following output:

```
Saab
```

Modify Values in Existing Arrays

To modify a value in an existing array, just add a new value to the array with a specified index number:

```
mycars[0]="Opel"
```

Now, the following code line:

```
document.write(mycars[0])
```

will result in the following output:

```
Opel
```

Two-dimensional array

The following code creates a two-dimensional array and displays the results.

```
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];
  }
document.write(str,"<p>")
}
```

This example displays the following results:

```
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]
```

```
arr2[2] = "Wendy";
document.write(arr.concat(arr2));
```

The output of the code above will be:

```
Jani,Tove,Hege,John,Andy,Wendy
```

```
document.write(arr.concat(arr2,arr3))
```

The output of the code above will be:

```
Jani,Tove,Hege,John,Andy,Wendy,Stale,Borge
```

eval(*string*)

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

- *string* is any string representing a JavaScript expression, statement, or sequence of statements. The expression can include variables and properties of existing objects.

join(*separator*)

Joins all elements of an array into a string.

The string conversion of all array elements are joined into one string.

- *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 (,).

Example:

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")
document.write(a.join() +"<br />")
document.write(a.join(", ") +"<br />")
document.write(a.join(" + ") +"<br />")
```

This code produces the following output:

```
Wind,Rain,Fire
Wind, Rain, Fire
Wind + Rain + Fire
```

pop()

Removes and returns the last element of an array

The pop() method is used to remove and return the last element of an array.

Note:

This method changes the length of the array.

Tip:

To remove and return the first element of an array, use the shift() method.

Example:

In this example we will create an array, and then remove the last element of the array. Note that this will also change the length of the array:

```
var arr = new Array(3)
arr[0] = "Jani"; arr[1] = "Hege"; arr[2] = "Stale"
document.write(arr + "<br />")
document.write(arr.pop() + "<br />")
document.write(arr)
```

The output of the code above will be:

```
Jani,Hege,Stale
Stale
Jani,Hege
```

push()

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

The **push()** method adds one or more elements to the end of an array and returns the new length.

Arguments

- *newelement1* (Required) The first element to add to the array
- *newelement2* (Optional) The second element to add to the array
- *newelementX* (Optional) Several elements may be added

Note:

This method changes the length of the array.

Tip:

To add one or more elements to the beginning of an array, use the unshift() method.

Example:

In this example we will create an array, and then change the length of it by adding a element:

```
var arr = new Array(3)
arr[0] = "Jani"; arr[1] = "Hege";
arr[2] = "Stale";
document.write(arr + "<br />");
document.write(arr.push("Kai Jim")+"<br/>");
document.write(arr)
```

The output of the code above will be:

```
Jani,Hege,Stale
4
Jani,Hege,Stale,Kai Jim
```

reverse()

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

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

Example:

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"*

shift()

Removes and returns the first element of an array

The **shift()** method is used to remove and return the first element of an array.

Note:

This method changes the length of the array.

Tip:

To remove and return the last element of an array, use the pop() method.

Example:

In this example we will create an array, and then remove the first element of the array. Note that this will also change the length of the array:

```
var arr = new Array(3);
arr[0] = "Jani"; arr[1] = "Hege"; arr[2] = "Stale";
document.write(arr + "<br />");
document.write(arr.shift() + "<br />");
document.write(arr);
```

The output of the code above will be:

```
Jani,Hege,Stale
Jani
Hege,Stale
```

slice(*start*,*end*)

Returns selected elements from an existing array

The **slice()** method returns selected elements from an existing array.

Arguments:

1. *start* (Required) Specify where to start the selection. Must be a number
2. *end* (Optional) Specify where to end the selection. Must be a number

Note:

If end is not specified, slice() selects all elements from the specified start position and to the end of the array.

Tip:

You can use negative numbers to select from the end of the array.

Example №1:

In this example we will create an array, and then display selected elements from it:

```
var arr = new Array(3);
arr[0] = "Jani";
arr[1] = "Hege";
arr[2] = "Stale";
document.write(arr + "<br />");
document.write(arr.slice(1)+"<br/>");
document.write(arr);
```

The output of the code above will be:

```
Jani,Hege,Stale
Hege,Stale
Jani,Hege,Stale
```

Example №2:

In this example we will create an array, and then display selected elements from it:

```
var arr = new Array(6);
arr[0] = "Jani"; arr[1] = "Hege";
arr[2] = "Stale"; arr[3] = "Kai Jim";
arr[4] = "Borge"; arr[5] = "Tove";
document.write(arr + "<br />");
document.write(arr.slice(2,4)+"<br/>");
document.write(arr);
```

The output of the code above will be:

```
Jani,Hege,Stale,Kai Jim,Borge,Tove
Stale,Kai Jim
Jani,Hege,Stale,Kai Jim,Borge,Tove
```

sort(*sortby*)

Sorts the elements of an array

The sort() method is used to sort the elements of an array.

Argument

- *sortby* (Optional) Specifies the sort order. Must be a function

Note:

- The sort() method will sort the elements alphabetically by default. However, this means that numbers will not be sorted correctly (40 comes before 5). To sort numbers, you must create a function that compare numbers.
- After using the sort() method, the array is changed.

Example №1:

In this example we will create an array and sort it alphabetically:

```
var arr = new Array(6);
arr[0] = "Jani";
arr[1] = "Hege";
arr[2] = "Stale";
arr[3] = "Kai Jim";
arr[4] = "Borge";
arr[5] = "Tove";
document.write(arr + "<br />");
document.write(arr.sort());
```

The output of the code above will be:

```
Jani,Hege,Stale,Kai Jim,Borge,Tove
Borge,Hege,Jani,Kai Jim,Stale,Tove
```

Example №2:

In this example we will create an array containing numbers and sort it:

```
var arr = new Array(6);
arr[0] = "10"; arr[1] = "5"; arr[2] = "40";
arr[3] = "25"; arr[4] = "1000"; arr[5] = "1";
document.write(arr + "<br />");
document.write(arr.sort());
```

The output of the code above will be:

```
10,5,40,25,1000,1
1,10,1000,25,40,5
```

Note

that the numbers above are NOT sorted correctly (by numeric value). To solve this problem, we must add a function that handles this problem:

Example №3:

```
function sortNumber(a,b) {return a-b}
var arr = new Array(6);
arr[0] = "10"; arr[1] = "5"; arr[2] = "40";
arr[3] = "25"; arr[4] = "1000"; arr[5] = "1";
document.write(arr + "<br />");
document.write(arr.sort(sortNumber));
```

The output of the code above will be:

```
10,5,40,25,1000,1
1,5,10,25,40,1000
```

splice (*index*,*howmany*,*element1*, ...,*elementX*)

Removes and adds new elements to an array

The splice() method is used to remove and add new elements to an array.

Arguments:

Example №1:

In this example we will create an array and add an element to it:

```
var arr = new Array(5);
arr[0] = "Jani"; arr[1] = "Hege";
arr[2] = "Stale"; arr[3] = "Kai Jim";
```

Example №3:

In this example we will remove three elements starting at index 2 ("Stale"), and add a new element ("Tove") there instead:

```
var arr = new Array(5);
arr[0] = "Jani";
```

- *index* (Required) Specify where to add/remove elements. Must be a number
- *howmany* (Required) Specify how many elements should be removed. Must be a number, but can be "0"
- *element1* (Optional) Specify a new element to add to the array
- *elementX* (Optional) Several elements can be added

```
arr[2] = "Stale"; arr[3] = "Kai Jim";
arr[4] = "Borge";
document.write(arr + "<br />");
arr.splice(2,0,"Lene");
document.write(arr + "<br />");
```

The output of the code above will be:

```
Jani,Hege,Stale,Kai Jim,Borge
Jani,Hege,Lene,Stale,Kai Jim,Borge
```

Example №2:
In this example we will remove the element at index 2 ("Stale"), and add a new element ("Tove") there instead:

```
var arr = new Array(5);
arr[0] = "Jani"; arr[1] = "Hege";
arr[2] = "Stale"; arr[3] = "Kai Jim";
arr[4] = "Borge";
document.write(arr + "<br />");
arr.splice(2,1,"Tove");
document.write(arr);
```

The output of the code above will be:

```
Jani,Hege,Stale,Kai Jim,Borge
Jani,Hege,Tove,Kai Jim,Borge
```

```
arr[0] = "Jani";
arr[1] = "Hege";
arr[2] = "Stale";
arr[3] = "Kai Jim";
arr[4] = "Borge";
document.write(arr + "<br />");
arr.splice(2,3,"Tove");
document.write(arr);
```

The output of the code above will be:

```
Jani,Hege,Stale,Kai Jim,Borge
Jani,Hege,Tove
```

unshift (*newelement1*,*newelement2*, ...,*newelementX*)

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

The unshift() method adds one or more elements to the beginning of an array and returns the new length.

Arguments:

- *newelement1* (Required) The first element to add to the array
- *newelement2* (Optional) The second element to add to the array
- *newelementX* (Optional) Several elements may be added

Note:

- This method changes the length of the array.
- The unshift() method does not work properly in Internet Explorer!

Tip:

- To add one or more elements to the end of an array, use the push() method.

Example
In this example we will create an array, add an element to the beginning of the array and then return the new length:

```
var arr = new Array();
arr[0] = "Jani"; arr[1] = "Hege";
arr[2] = "Stale";
document.write(arr + "<br />");
document.write(arr.unshift("Kai Jim")+"<br/>");
document.write(arr);
```

The output of the code above will be:

```
Jani,Hege,Stale
4
Kai Jim,Jani,Hege,Stale
```

JavaScript Boolean Object Description

Review

Use the built-in 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.

To create a Boolean object:

```
var booleanObjectName = new Boolean(value)
```

- *booleanObjectName* is either the name of a new object or a property of an existing object. When using Boolean properties, *booleanObjectName* is either the name of an existing Boolean object or a property of an existing object.
- *value* is 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 "", it the object has an initial value of false. All other values, including the string "false" create an object with an initial value of true.

The following examples create Boolean objects:

```
bfalse = new Boolean(false)
btrue = new Boolean(true)
```

All the following lines of code create Boolean objects with an initial value of false:

```
var myBoolean=new Boolean(); var myBoolean=new Boolean(0); var myBoolean=new Boolean(null)
var myBoolean=new Boolean(""); var myBoolean=new Boolean(false); var myBoolean=new Boolean(NaN)
```

And all the following lines of code create Boolean objects with an initial value of true:

```
var myBoolean=new Boolean(true); var myBoolean=new Boolean("true")
var myBoolean=new Boolean("false"); var myBoolean=new Boolean("Richard")
```

JavaScript Data Object Description

Review

JavaScript does not have a date data type. However, you can use the Date object and its methods to work with dates and times in your applications. The Date object has a large number of methods for setting, getting, and manipulating dates. It does not have any properties.

JavaScript handles dates similarly to Java. The two languages have many of the same date methods, and both languages store dates as the number of milliseconds since January 1, 1970, 00:00:00.

Note

Currently, you cannot work with dates prior to January 1, 1970.

To create a Date object:

```
dateObjectName = new Date([parameters])
```

where *dateObjectName* is the name of the Date object being created; it can be a new object or a property of an existing object.

The parameters in the preceding syntax can be any of the following:

- Nothing: creates today's date and time. For example, *today = new Date()*.
- A string representing a date in the following form: "Month day, year hours:minutes:seconds." For example, *Xmas95 = new Date("December 25, 1995 13:30:00")*. If you omit hours, minutes, or seconds, the value will be set to zero.
- A set of integer values for year, month, and day. For example, *Xmas95 = new Date(95,11,25)*. A set of values for year, month, day, hour, minute, and seconds. For example, Xmas95 = new Date(95,11,25,9,30,0)

Methods of the Date object

The Date object methods for handling dates and times fall into these broad categories:

- "set" methods, for setting date and time values in Date objects.
- "get" methods, for getting date and time values from Date objects.
- "to" methods, for returning string values from Date objects.
- *parse* and *UTC* methods, for parsing Date strings.

With the "get" and "set" methods you can get and set seconds, minutes, hours, day of the month, day of the week, months, and years separately. There is a getDay method that returns the day of the week, but no corresponding setDay method, because the day of the week is set automatically.

These methods use integers to represent these values as follows:

- Seconds and minutes: 0 to 59
- Hours: 0 to 23
- Day: 0 to 6 (day of the week)
- Date: 1 to 31 (day of the month)
- Months: 0 (January) to 11 (December)
- Year: years since 1900

For example, suppose you define the following date:

```
Xmas95 = new Date("December 25, 1995")
```

Then Xmas95.getMonth() returns 11, and Xmas95.getYear() returns 95.

The getTime and setTime methods are useful for comparing dates. The getTime method returns the number of milliseconds since the epoch for a Date object.

For example, the following code displays the number of days left in the current year:

```
today = new Date()
endYear = new Date("December 31, 1990") // Set day and month
endYear.setYear(today.getYear()) // Set year to this year
msPerDay = 24 * 60 * 60 * 1000 // Number of millseconds per day
daysLeft = (endYear.getTime() - today.getTime()) / msPerDay
daysLeft = Math.round(daysLeft)
document.write("Number of days left in the year: " + daysLeft)
```

This example creates a Date object named today that contains today's date. It then creates a Date object named endYear and sets the year to the current year.

JavaScript Data Object Methods

Date()

Returns today's date and time

Example
In this example we print the day of the current month:

```
document.write(Date())
```

The output of the code above will be:

```
Sat Mar 04 2017 11:37:01 GMT-0600 (CST)
```

getDate()

Returns the day of the month from a Date object (from 1-31)

Note:

- The value returned by getDate() is a number between 1 and 31.
- This method is always used in conjunction with a Date object.

Example №1
In this example we print the day of the current month:

```
var d = new Date();
document.write(d.getDate())
```

The output of the code above will be:

```
4
```

Example №2
Here we define a variable with a specific date and then print the day of the month in the variable:

```
var birthday = new Date("July 21, 1983 01:15:00");
document.write(birthday.getDate())
```

The output of the code above will be:

```
21
```

getDay()

Returns the day of the week from a Date object (from 0-6)

Note:

- The value returned by getDay() is a number between 0 and 6. Sunday is 0, Monday is 1 and so on.
- This method is always used in conjunction with a Date object.

Example №1
In this example we get the current day (as a number) of the week:

```
var d = new Date();
document.write(d.getDay())
```

The output of the code above will be:

```
6
```

Example №2
Now we will create an array to get our example to write a weekday, and not just a number:

```
var d=new Date(); var weekday=new Array(7);
weekday[0]="Sunday"; weekday[1]="Monday";
weekday[2]="Tuesday"; weekday[3]="Wednesday";
weekday[4]="Thursday"; weekday[5]="Friday";
weekday[6]="Saturday";
document.write("Today it is " + weekday[d.getDay()]);
```

The output of the code above will be:

```
Today it is Saturday
```

getMonth()

Returns the month from a Date object (from 0-11)

Note:

- The value returned by getMonth() is a number between 0 and 11. January is 0, February is 1 and so on.
- This method is always used in conjunction with a Date object.

Example №1
In this example we get the current month and print it:

```
var d = new Date();
document.write(d.getMonth())
```

The output of the code above will be:

```
2
```

Example №2
Now we will create an array to get our example to write the name of the month, and not just a number:

```
var d=new Date(); var month=new Array(12);
month[0]="January"; month[1]="February";
month[2]="March"; month[3]="April";
month[4]="May"; month[5]="June";
month[6]="July"; month[7]="August";
month[8]="September"; month[9]="October";
month[10]="November"; month[11]="December";
document.write("The month is "+month[d.getMonth()]);
```

The output of the code above will be:

```
The month is March
```

getFullYear()

Returns the year, as a four-digit number, from a Date object

Note:

- This method is always used in conjunction with a Date object.

Example №1
In this example we get the current year and print it:

```
var d = new Date();
document.write(d.getFullYear())
```

The output of the code above will be:

```
2017
```

Example №2
Here we will extract the year out of the specific date:

```
var born = new Date("July 21, 1983 01:15:00");
document.write("I was born in " + born.getFullYear())
```

The output of the code above will be:

```
I was born in 1983
```

getHours()

Returns the hour of a Date object (from 0-23)

Note:

- The value returned by getHours() is a two digit number. However, the return value is not always two digits, if the value is less than 10 it only returns one digit.
- This method is always used in conjunction with a Date object.

Example №1
In this example we get the hour of the current time:

```
var d = new Date();
document.write(d.getHours())
```

The output of the code above will be:

```
11
```

Example №2
Here we will extract the hour from the specific date and time:

```
var born = new Date("July 21, 1983 01:15:00");
document.write(born.getHours())
```

The output of the code above will be:

```
1
```

Then, using the number of milliseconds per day, it computes the number of days between today and endYear, using getTime and rounding to a whole number of days.

The parse method is useful for assigning values from date strings to existing Date objects. For example, the following code uses parse and setTime to assign a date value to the IPOdate object:

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

Using the Date object: an example

The following example shows a simple application of Date: it displays a continuously-updated digital clock in an HTML text field. This is possible because you can dynamically change the contents of a text field with JavaScript (in contrast to ordinary text, which you cannot update without reloading the document).

The display in Navigator looks like this:

The current time is 11:37:30 A.M.

The <body>: of the document is:

```
<body onLoad="JSClock()">
  <form name="clockForm">
    The current time is
    <input type="text" name="digits" size="12" value="" />
  </form>
</body>
```

The <body> tag includes an onLoad event handler. When the page loads, the event handler calls the function JSClock, defined in the <head>. A form called clockForm includes a single text field named digits, whose value is initially an empty string.

The <head> of the document defines JSClock as follows:

```
<head>
  <script type="text/javascript">
    <!--
    function JSClock() {
      var time = new Date()
      var hour = time.getHours()
      var minute = time.getMinutes()
      var second = time.getSeconds()
      var temp = "" + ((hour > 12) ? hour - 12 : hour)
      temp += ((minute < 10) ? ":0" : ":") + minute
      temp += ((second < 10) ? ":0" : ":") + second
      temp += (hour >= 12) ? " P.M." : " A.M."
      document.clockForm.digits.value = temp
      id = setTimeout("JSClock()",1000)
    }
    //-->
  </script>
</head>
```

The JSClock function first creates a new Date object called time; since no arguments are given, time is created with the current date and time. Then calls to the getHours, getMinutes, and getSeconds methods assign the value of the current hour, minute and seconds to hour, minute, and second.

The next four statements build a string value based on the time. The first statement creates a variable temp, assigning it a value using a conditional expression; if hour is greater than 12, (hour - 13), otherwise simply hour.

The next statement appends a minute value to temp. If the value of minute is less than 10, the conditional expression adds a string with a preceding zero; otherwise it adds a string with a demarcating colon. Then a statement appends a seconds value to temp in the same way.

Finally, a conditional expression appends "PM" to temp if hour is 12 or greater; otherwise, it appends "AM" to temp.

The next statement assigns the value of temp to the text field:

```
document.aform.digits.value = temp
```

This displays the time string in the document.

The final statement in the function is a recursive call to JSClock:

```
id = setTimeout("JSClock()", 1000)
```

The built-in JavaScript setTimeout function specifies a time delay to evaluate an expression, in this case a call to JSClock. The second argument indicates a a delay of 1,000 milliseconds (one second). This updates the display of time in the form at one-second intervals.

Note that the function returns a value (assigned to id), used only as an identifier (which can be used by the clearTimeout method to cancel the evaluation).

Manipulate Dates

We can easily manipulate the date by using the methods available for the Date object.

In the example below we set a Date object to a specific date (14th January 2010):

```
var myDate=new Date()
myDate.setFullYear(2010,0,14)
```

And in the following example we set a Date object to be 5 days into the future:

```
var myDate=new Date()
myDate.setDate(myDate.getDate()+5)
```

Note:

If adding five days to a date shifts the month or year, the changes are handled automatically by the Date object itself!

Comparing Dates

The Date object is also used to compare two dates.

The following example compares today's date with the 14th January 2010:

```
var myDate=new Date()
myDate.setFullYear(2010,0,14)
var today = new Date()
if (myDate>today)
  alert("Today is before 14th January 2010")
else
  alert("Today is after 14th January 2010")
```

getMinutes()

Returns the minutes of a Date object (from 0-59)

Note:

- The value returned by getMinutes() is a two digit number. However, the return value is not always two digits, if the value is less than 10 it only returns one digit.
- This method is always used in conjunction with a Date object.

Example №1

In this example we get the minutes of the current time:

```
var d = new Date();
document.write(d.getMinutes())
```

The output of the code above will be:

```
37
```

Example №2

Here we will extract the minutes from the specific date and time:

```
var born = new Date("July 21, 1983 01:15:00");
document.write(born.getMinutes())
```

The output of the code above will be:

```
15
```

getSeconds()

Returns the seconds of a Date object (from 0-59)

Note:

- The value returned by getSeconds() is a two digit number. However, the return value is not always two digits, if the value is less than 10 it only returns one digit.
- This method is always used in conjunction with a Date object.

Example №1

In this example we get the seconds of the current time:

```
var d = new Date();
document.write(d.getSeconds())
```

The output of the code above will be:

```
1
```

Example №2

Here we will extract the seconds from the specific date and time:

```
var born = new Date("July 21, 1983 01:15:00");
document.write(born.getSeconds())
```

The output of the code above will be:

```
0
```

getMilliseconds()

Returns the milliseconds of a Date object (from 0-999)

Note:

- The value returned by getMilliseconds() is a three digit number. However, the return value is not always three digits, if the value is less than 100 it only returns two digits, and if the value is less than 10 it only returns one digit.
- This method is always used in conjunction with a Date object.

Example №1

In this example we get the milliseconds of the current time:

```
var d = new Date();
document.write(d.getMilliseconds())
```

The output of the code above will be:

```
415
```

Example №2

Here we will extract the milliseconds from the specific date and time:

```
var born = new Date("July 21, 1983 01:15:00");
document.write(born.getMilliseconds())
```

The output of the code above will be:

```
0
```

getTime()

Returns the number of milliseconds since midnight Jan 1, 1970

Note:

- This method is always used in conjunction with a Date object.

Example №1

In this example we will get how many milliseconds since 1970/01/01 and print it:

```
var d = new Date();
document.write(d.getTime() + " milliseconds since
1970/01/01")
```

The output of the code above will be:

```
1488649021416 milliseconds since 1970/01/01
```

Example №2

In the following example we will calculate the number of years since 1970/01/01:

```
var minutes = 1000*60; var hours = minutes*60;
var days = hours*24; var years = days*365;
var d = new Date(); var t=d.getTime(); var y=t/years;
document.write("It's been: "+y+" years since
1970/01/01!");
```

The output of the code above will be:

```
It's been: 47.2047508062214 6 years since 1970/01/01!
```

getTimezoneOffset()

Returns the difference in minutes between local time and Greenwich Mean Time (GMT)

Note:

- The returned value of this method is not a constant, because of the practice of using Daylight Saving Time.
- This method is always used in conjunction with a Date object.

Example №1

In the following example we get the difference in minutes between local time and Greenwich Mean Time (GMT):

```
var d = new Date();
document.write(d.getTimezoneOffset())
```

The output of the code above will be:

```
360
```

Example №2

Now we will convert the example above into GMT +/- hours:

```
var d = new Date();
var gmtHours = d.getTimezoneOffset()/60;
document.write("The local time zone is: GMT " +
gmtHours);
```

The output of the code above will be:

```
The local time zone is: GMT 6
```

getUTCDate()

Returns the day of the month from a Date object according to universal time (from 1-31)

Note:

- The value returned by getUTCDate() is a number between 1 and 31.
- This method is always used in conjunction with a Date object.

Tip: The Universal Coordinated Time (UTC) is the time set by the World Time Standard.

Example №1

In this example we print the current day of the month according to UTC:

```
var d = new Date();
document.write(d.getUTCDate())
```

The output of the code above will be:

```
4
```

Example №2

Here we define a variable with a specific date and then print the day of the month in the variable, according to UTC:

```
var born = new Date("July 21, 1983 01:15:00");
document.write(born.getUTCDate())
```

The output of the code above will be:

```
21
```

getUTCDay()

Returns the day of the week from a Date object according to universal time (from 0-6)

Note:

- The value returned by getUTCDay() is a number between 0 and 6. Sunday is 0, Monday is 1 and so on.
- This method is always used in conjunction with a Date object.

Tip: The Universal Coordinated Time (UTC) is the time set by the World Time Standard.

Example №1

In this example we get the current UTC day (as a number) of the week:

```
var d = new Date();
document.write(d.getUTCDay())
```

The output of the code above will be:

```
6
```

Example №2

Now we will create an array to get our example above to write a weekday, and not just a number:

```
var d=new Date(); var weekday=new Array(7);
weekday[0]="Sunday"; weekday[1]="Monday";
weekday[2]="Tuesday"; weekday[3]="Wednesday";
weekday[4]="Thursday"; weekday[5]="Friday";

weekday[6]="Saturday";
document.write("Today it is "+weekday[d.getUTCDay()])
```

The output of the code above will be:

```
Today it is Saturday
```

getUTCMonth()

Returns the month from a Date object according to universal time (from 0-11)

Note:

- The value returned by getUTCMonth() is a number between 0 and 11. January is 0, February is 1 and so on.
- This method is always used in conjunction with a Date object.

Tip: The Universal Coordinated Time (UTC) is the time set by the World Time Standard.

Example №1

In this example we get the current month and print it:

```
var d = new Date();
document.write(d.getUTCMonth())
```

The output of the code above will be:

```
2
```

Example №2

Now we will create an array to get our example to write the name of the month, and not just a number:

```
var d=new Date(); var month=new Array(12);
month[0]="January"; month[1]="February";
month[2]="March"; month[3]="April";
month[4]="May"; month[5]="June";
month[6]="July"; month[7]="August";
month[8]="September"; month[9]="October";
month[10]="November"; month[11]="December";
document.write("The month is "+month[d.getUTCMonth()]);
```

The output of the code above will be:

```
The month is March
```

getUTCFullYear()

Returns the four-digit year from a Date object according to universal time

Note:

- This method is always used in conjunction with a Date object.

Tip: The Universal Coordinated Time (UTC) is the time set by the World Time Standard.

Example №1

In this example we get the current year and print it:

```
var d = new Date();
document.write(d.getUTCHours())
```

The output of the code above will be:

```
2017
```

Example №2

Here we will extract the year out of the specific date:

```
var born = new Date("July 21, 1983 01:15:00");
document.write("I was born in " + born.getUTCFullYear())
```

The output of the code above will be:

```
I was born in 1983
```

getUTCHours()

Returns the hour of a Date object according to universal time (from 0-23)

Note:

- The value returned by getUTCHours() is a two digit number. However, the return value is not always two digits, if the value is less than 10 it only returns one digit.
- This method is always used in conjunction with a Date object.

Tip: The Universal Coordinated Time (UTC) is the time set by the World Time Standard.

Example №1

In this example we get the UTC hour of the current time:

```
var d = new Date();
document.write(d.getUTCHours())
```

The output of the code above will be:

```
17
```

Example №2

Here we will extract the UTC hour from the specific date and time:

```
var born = new Date("July 21, 1983 01:15:00");
document.write(born.getUTCHours())
```

The output of the code above will be:

```
6
```

getUTCMinutes()

Returns the minutes of a Date object according to universal time (from 0-59)

Note:

- The value returned by getUTCMinutes() is a two digit number. However, the return value is not always two digits, if the value is less than 10 it only returns one digit.
- This method is always used in conjunction with a Date object.

Tip: The Universal Coordinated Time (UTC) is the time set by the World Time Standard.

Example №1

In this example we get the UTC minutes of the current time:

```
var d = new Date();
document.write(d.getUTCMinutes())
```

The output of the code above will be:

```
37
```

Example №2

Here we will extract the UTC minutes from the specific date and time:

```
var born = new Date("July 21, 1983 01:15:00");
document.write(born.getUTCMinutes())
```

The output of the code above will be:

```
15
```

getUTCSeconds()

Returns the seconds of a Date object

getUTCSeconds()

Returns the seconds of a Date object according to universal time (from 0-59)

Note:

- The value returned by getUTCSeconds() is a two digit number. However, the return value is not always two digits, if the value is less than 10 it only returns one digit.
- This method is always used in conjunction with a Date object.

Tip: The Universal Coordinated Time (UTC) is the time set by the World Time Standard.

Example №1

In this example we get the UTC seconds of the current time:

```
var d = new Date();
document.write(d.getUTCSeconds())
```

The output of the code above will be:

```
1
```

Example №2

Here we will extract the UTC seconds from the specific date and time:

```
var born = new Date("July 21, 1983 01:15:00");
document.write(born.getUTCSeconds())
```

The output of the code above will be:

```
0
```

getUTCMilliseconds()

Returns the milliseconds of a Date object according to universal time (from 0-999)

Note:

- The value returned by getUTCMilliseconds() is a three digit number. However, the return value is not always three digits, if the value is less than 100 it only returns two digits, and if the value is less than 10 it only returns one digit.
- This method is always used in conjunction with a Date object.

Tip: The Universal Coordinated Time (UTC) is the time set by the World Time Standard.

Example №1

In this example we get the UTC milliseconds of the current time:

```
var d = new Date();
document.write(d.getUTCMilliseconds())
```

The output of the code above will be:

```
577
```

Example №2

Here we will extract the UTC milliseconds from the specific date and time:

```
var born = new Date("July 21, 1983 01:15:00");
document.write(born.getUTCMilliseconds())
```

The output of the code above will be:

```
0
```

Note: The code above will set the milliseconds to 0, since no milliseconds was defined in the date.

parse(*datestring*)

Takes a date string and returns the number of milliseconds since midnight of January 1, 1970

Argument:

- *datestring* (Required) A string representing a date

Example №1

In this example we will get how many milliseconds there are from 1970/01/01 to 2005/07/08:

```
var d = Date.parse("Jul 8, 2005");
document.write(d)
```

The output of the code above will be:

```
1120798800000
```

Example №2

Now we will convert the output from the example above into years:

```
var minutes = 1000 * 60; var hours = minutes * 60;
var days = hours * 24; var years = days * 365;
var t = Date.parse("Jul 8, 2005");
var y = t/years;
document.write("It's been: " + y + " years from
1970/01/01!");
document.write(" to 2005/07/08!")
```

The output of the code above will be:

```
It's been: 35.54029680365297 years from 1970/01/01 to
2005/07/08!
```

setDate()

Sets the day of the month in a Date object (from 1-31)

Arguments:

- *day* (Required) A numeric value (from 1 to 31) that represents a day in a month

Note:

- This method is always used in conjunction with a Date object.

Example №

In this example we set the day of the current month to 15 with the setDate() method:

```
var d = new Date();
d.setDate(15);
document.write(d);
```

The output of the code above will be:

```
Wed Mar 15 2017 11:37:01 GMT-0500 (CDT)
```

setMonth(*month*,*day*)

Sets the month in a Date object (from 0-11)

Arguments:

- *month* (Required) A numeric value between 0 and 11 representing the month
- *day* (Optional) A numeric value between 1 and 31 representing the date

Note:

- The value set by setMonth() is a number between 0 and 11. January is 0, February is 1 and so on.
- This method is always used in conjunction with a Date object.

Example №1

In this example we set the month to 0 (January) with the setMonth() method:

```
var d=new Date();
d.setMonth(0);
document.write(d)
```

The output of the code above will be:

```
Wed Jan 04 2017 11:37:01 GMT-0600 (CST)
```

Example №2

In this example we set the month to 0 (January) and the day to 20 with the setMonth() method:

```
var d=new Date();
d.setMonth(0,20);
document.write(d)
```

The output of the code above will be:

```
Fri Jan 20 2017 11:37:01 GMT-0600 (CST)
```

setFullYear(*year*,*month*,*day*)

Sets the year in a Date object (four digits)

Arguments:

- *year* (Required) A four-digit value representing the year
- *month* (Optional) A numeric value between 0 and 11 representing the month
- *day* (Optional) A numeric value between 1 and 31 representing the date

Note: This method is always used in conjunction with a Date object.

Example №1

In this example we set the year to 1992 with the setFullYear() method:

```
var d = new Date();
d.setFullYear(1992);
document.write(d)
```

The output of the code above will be:

```
Wed Mar 04 1992 11:37:01 GMT-0600 (CST)
```

Example №2

In this example we set the date to November 3, 1992 with the setFullYear() method:

```
var d = new Date();
d.setFullYear(1992,10,3);
document.write(d)
```

The output of the code above will be:

```
Tue Nov 03 1992 11:37:01 GMT-0600 (CST)
```

setHours(*hour*,*min*,*sec*,*millisec*)

Sets the hour in a Date object (from 0-23)

Argument:

- *hour* (Required) A numeric value between 0 and 23 representing the hour
- *min* (Required) A numeric value between 0 and 59 representing the minutes
- *sec* (Optional) A numeric value between 0 and 59 representing the seconds
- *millisec* (Optional) A numeric value between 0 and 999 representing the milliseconds

Note:

- If one of the parameters above is specified with a one-digit number, JavaScript adds one or two leading zeros in the result.
- This method is always used in conjunction with a Date object.

Example

In this example we set the hour of the current time to 15, with the setHours() method:

```
var d = new Date();
d.setHours(15);

document.write(d)
```

The output of the code above will be:

```
Sat Mar 04 2017 15:37:01 GMT-0600 (CST)
```

setMinutes(*min*,*sec*,*millisec*)

Set the minutes in a Date object (from 0-59)

Argument:

- *min* (Required) A numeric value between 0 and 59 representing the minutes
- *sec* (Optional) A numeric value between 0 and 59 representing the seconds
- *millisec* (Optional) A numeric value between 0 and 999 representing the milliseconds

Note:

- If one of the parameters above is specified with a one-digit number, JavaScript adds one or two leading zeros in the result.
- This method is always used in conjunction with a Date object.

Example

In this example we set the minutes of the current time to 01, with the setMinutes() method:

```
var d = new Date();
d.setMinutes(1);
document.write(d)
```

The output of the code above will be:

```
Sat Mar 04 2017 11:01:01 GMT-0600 (CST)
```

setSeconds(*sec*,*millisec*)

Sets the seconds in a Date object (from 0-59)

Argument:

- *sec* (Required) A numeric value between 0 and 59 representing the seconds
- *millisec* (Optional) A numeric value between 0 and 999 representing the milliseconds

Note:

- If one of the parameters above is specified with a one-digit number, JavaScript adds one or two leading zeros in the result.
- This method is always used in conjunction with a Date object.

Example

In this example we set the seconds of the current time to 01, with the setSeconds() method:

```
var d = new Date();
d.setSeconds(1);
document.write(d)
```

The output of the code above will be:

```
Sat Mar 04 2017 11:37:01 GMT-0600 (CST)
```

setMilliseconds(*millisec*)

Sets the milliseconds in a Date object (from 0-999)

Argument:

- *millisec* (Required) A numeric value between 0 and 999 representing the milliseconds

Note:

- If the parameter above is specified with a one-digit or two-digit number, JavaScript adds one or two leading zeros in the result.
- This method is always used in conjunction with a Date object.

Example

In this example we set the milliseconds of the current time to 001, with the setMilliseconds() method:

```
var d = new Date();
d.setMilliseconds(1);
document.write(d)
```

The output of the code above will be:

```
Sat Mar 04 2017 11:37:01 GMT-0600 (CST)
```

setTime(*millisec*)

Calculates a date and time by adding or subtracting a specified number of milliseconds to/from midnight January 1, 1970

Argument:

Example №1

In this example we will add 77771564221 milliseconds to 1970/01/01 and display the new date and time:

Example №2

In this example we will subtract 77771564221 milliseconds from 1970/01/01 and display the new date and time:

- *millisec* (Required) A numeric value representing the milliseconds since midnight January 1, 1970. Can be a negative number

Note: This method is always used in conjunction with a Date object.

var d = new Date();
d.setTime(77771564221);
document.write(d);

The output of the code above will be:

Sun Jun 18 1972 22:12:44 GMT-0500 (CDT)

var d = new Date();
d.setTime(-77771564221);
document.write(d)

The output of the code above will be:

Sat Jul 15 1967 15:47:15 GMT-0500 (CDT)

setUTCDate(*day*)

Sets the day of the month in a Date object according to universal time (from 1-31)

Argument:

- *day* (Required) A numeric value between 1 and 31 representing the date

Note: This method is always used in conjunction with a Date object.
Tip: The Universal Coordinated Time (UTC) is the time set by the World Time Standard.

Example

In this example we set the day of the current month to 15 with the setUTCDate() method:

var d = new Date();
d.setUTCDate(15);
document.write(d)

The output of the code above will be:

Wed Mar 15 2017 12:37:01 GMT-0500 (CDT)

setUTCMonth(*month*,*day*)

Sets the month in a Date object according to universal time (from 0-11)

Arguments:

- *month* (Required) A numeric value between 0 and 11 representing the month
- *day* (Optional) A numeric value between 1 and 31 representing the date

Note: This method is always used in conjunction with a Date object.
Tip: The Universal Coordinated Time (UTC) is the time set by the World Time Standard.

Example №1

In this example we set the month to 0 (January) with the setUTCMonth() method:

var d = new Date();
d.setUTCMonth0);
document.write(d)

The output of the code above will be:

Wed Jan 04 2017 11:37:01 GMT-0600 (CST)

Example №2

In this example we set the month to 0 (January) and the day to 20 with the setUTCMonth() method:

var d = new Date();
d.setUTCMonth(0,20);
document.write(d)

The output of the code above will be:

Fri Jan 20 2017 11:37:01 GMT-0600 (CST)

setUTCFullYear(*year*,*month*,*day*)

Sets the year in a Date object according to universal time (four digits)

Arguments:

- *year* (Required) A four-digit value representing the year
- *month* (Optional) A numeric value between 0 and 11 representing the month
- *day* (Optional) A numeric value between 1 and 31 representing the date

Note: This method is always used in conjunction with a Date object.
Tip: The Universal Coordinated Time (UTC) is the time set by the World Time Standard.

Example №1

In this example we set the year to 1992 with the setUTCFullYear() method:

var d = new Date();
d.setUTCFullYear(1992);
document.write(d)

The output of the code above will be:

Wed Mar 04 1992 11:37:01 GMT-0600 (CST)

Example №2

In this example we set the date to November 3, 1992 with the setUTCFullYear() method:

var d = new Date();
d.setUTCFullYear(1992,10,3);
document.write(d)

The output of the code above will be:

Tue Nov 03 1992 11:37:01 GMT-0600 (CST)

setUTCHours (*hour*,*min*,*sec*,*millisec*)

Sets the hour in a Date object according to universal time (from 0-23)

Arguments:

- *hour* (Required) A numeric value between 0 and 23 representing the hour
- *min* (Optional) A numeric value between 0 and 59 representing the minutes
- *sec* (Optional) A numeric value between 0 and 59 representing the seconds
- *millisec* (Optional) A numeric value between 0 and 999 representing the milliseconds

Note:

- If one of the parameters above is specified with a one-digit number, JavaScript adds one or two leading zeros in the result.
- This method is always used in conjunction with a Date object.

Tip: The Universal Coordinated Time (UTC) is the time set by the World Time Standard.

Example

In this example we set the seconds of the current time to 01, with the setUTCSeconds() method:

var d = new Date();
d.setUTCHours(1);
document.write(d)

The output of the code above will be:

Fri Mar 03 2017 19:37:01 GMT-0600 (CST)

setUTCMinutes(*min*,*sec*,*millisec*)

Set the minutes in a Date object according to universal time (from 0-59)

Arguments:

- *min* (Required) A numeric value between 0 and 59 representing the minutes
- *sec* (Optional) A numeric value between 0 and 59 representing the seconds
- *millisec* (Optional) A numeric value between 0 and 999 representing the milliseconds

Note:

- If one of the parameters above is specified with a one-digit number, JavaScript adds one or two leading zeros in the result.
- This method is always used in conjunction with a Date object.

Tip: The Universal Coordinated Time (UTC) is the time set by the World Time Standard.

Example

In this example we set the seconds of the current time to 01, with the setUTCSeconds() method:

var d = new Date();
d.setUTCMinutes(1);
document.write(d)

The output of the code above will be:

Sat Mar 04 2017 11:01:01 GMT-0600 (CST)

setUTCSeconds(*sec*,*millisec*)

Set the seconds in a Date object according to universal time (from 0-59)

Arguments:

- *sec* (Required) A numeric value between 0 and 59 representing the seconds
- *millisec* (Optional) A numeric value between 0 and 999 representing the milliseconds

Note:

- If one of the parameters above is specified with a one-digit number, JavaScript adds one or two leading zeros in the result.
- This method is always used in conjunction with a Date object.

Tip: The Universal Coordinated Time (UTC) is the time set by the World Time Standard.

Example

In this example we set the seconds of the current time to 01, with the setUTCSeconds() method:

var d = new Date();
d.setUTCSeconds(1);
document.write(d)

The output of the code above will be:

Sat Mar 04 2017 11:37:01 GMT-0600 (CST)

setUTCMilliseconds(*millisec*)

Sets the milliseconds in a Date object according to universal time (from 0-999)
Argument:

- *millisec* (Required) A numeric value between 0 and 999 representing the milliseconds

Note

- If the parameter above is specified with a one-digit or two-digit number, JavaScript adds one or two leading zeros in the result.
- This method is always used in conjunction with a Date object.

Tip: The Universal Coordinated Time (UTC) is the time set by the World Time Standard.

Example

In this example we set the milliseconds of the current time to 001, with the setUTCMilliseconds() method:

var d = new Date();
d.setUTCMilliseconds(1);
document.write(d)

The output of the code above will be:

Sat Mar 04 2017 11:37:01 GMT-0600 (CST)

toUTCString()

Converts a Date object, according to universal time, to a string

Example №1

In the example below we will convert today's date (according to UTC) to a string:

var d = new Date();
document.write (d.toUTCString())

The output of the code above will be:

Sat, 04 Mar 2017 17:37:01 GMT

Example №2

In the example below we will convert a specific date (according to UTC) to a string:

var born = new Date("December 29, 1970 00:30:00");
document.write(born.toUTCString())

The output of the code above will be:

Tue, 29 Dec 1970 06:30:00 GMT

toLocaleString()

Converts a Date object, according to local time, to a string

Example №1

In the example below we will convert today's date (according to local time) to a string:

var d = new Date();
document.write(d.toLocaleString())

The output of the code above will be:

3/4/2017, 11:37:01 AM

Example №2

In the example below we will convert a specific date (according to local time) to a string:

var born = new Date("December 29, 1970 00:30:00");
document.write(born.toLocaleString())

The output of the code above will be:

12/29/1970, 12:30:00 AM

UTC (*year*,*month*,*day*, *hours*,*minutes*,*seconds*,*ms*)

Takes a date and returns the number of milliseconds since midnight of January 1, 1970 according to universal time

Arguments

- *year* (Required) A four digit number representing the year
- *month* (Required) An integer between 0 and 11 representing the month
- *day* (Required) An integer between 1 and 31 representing the date
- *hours* (Optional) An integer between 0 and 23 representing the hour
- *minutes* (Optional) An integer between 0 and 59 representing the minutes
- *seconds* (Optional) An integer between 0 and 59 representing the seconds
- *ms* (Optional) An integer between 0 and 999 representing the milliseconds

Example №1

In this example we will get how many milliseconds there are from 1970/01/01 to 2005/07/08 according to universal time:

var d = Date.UTC(2005,7,8);
document.write(d)

The output of the code above will be:

1123459200000

Example №2

Now we will convert the output from the example above into years:

*var minutes = 1000*60;*
*var hours = minutes*60;*
*var days = hours*24;*
*var years = days*365;*
var t = Date.UTC(2005,7,8);
var y = t/years;
document.write("It's been: " + y + " years from 1970/01/01");
document.write(" to 2005/07/08!")

The output of the code above will be:

It's been: 35.62465753424657 years from 1970/01/01 to 2005/07/08!

JavaScript Function Object Description

The built-in Function object specifies a string of JavaScript code to be compiled as a function.
To create a Function object:

```
functionObjectName = new Function ([arg1, arg2, ... argn], functionBody)
```

Arguments:

- *functionObjectName* is the name of a variable or a property of an existing object. It can also be an object followed by a lowercase event handler name, such as window.onerror. When using Function properties, *functionObjectName* is either the name of an existing Function object or a property of an existing object.
- *arg1*, *arg2*, ... *argn* are arguments 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* is a string specifying the JavaScript code to be compiled as the function body.

Function objects 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.
In addition to defining functions as described here, you can also use the function statement, as described in "function".
The following code assigns a function to the variable setBGColor. This function sets the current document's background color.

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

To call the Function object, you can specify the variable name as if it were a function. The following code executes the function specified by the setBGColor variable:

```
var colorChoice="antiquewhite"
if (colorChoice=="antiquewhite") {setBGColor()}
```

You can assign the function to an event handler in either of the following ways:

```
1. document.form1.colorButton.onclick=setBGColor
2. <input name="colorButton" type="button" value="Change background color" onClick="setBGColor()">
```

Creating the variable setBGColor shown above is similar to declaring the following function:

```
function setBGColor() {
  document.bgColor='antiquewhite'
}
```

Assigning a function to a variable is similar to declaring a function, but they have differences:

- When you assign a function to a variable using var setBGColor = new Function("..."), setBGColor 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 setBGColor() {...}, setBGColor is not a variable, it is the name of a function.

JavaScript Image Object Description

To create an Image object:

```
imageName = new Image([width, height])
```

To use an Image object's properties:

```
1. imageName.propertyName
2. document.images[index].propertyName
3. formName.elements[index].propertyName
```

To define an event handler for an Image object created with the Image() constructor:

```
1. imageName.onabort = handlerFunction
2. imageName.onerror = handlerFunction
3. imageName.onload = handlerFunction
```

The position and size of an image in a document are set when the document is displayed in Navigator and cannot be changed using JavaScript (the width and height properties are read-only). You can change which image is displayed by setting the src and lowsrc properties. (See the descriptions of src and lowsrc.)

You can use JavaScript to create an animation with an Image object by repeatedly setting the src property. 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).
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 the Link object.

The Image() constructor
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 prefetched 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.

JavaScript Image Object Properties

border	Reflects the BORDER attribute
complete	Boolean value indicating whether Navigator 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
src	Reflects the SRC attribute
vspace	Reflects the VSPACE attribute
width	Reflects the WIDTH attribute

JavaScript Math Object Description

The built-in Math object has properties and methods for mathematical constants and functions. For example, the Math object's PI property has the value of pi (3.141...), which you would use in an application as

```
Math.PI
```

Similarly, standard mathematical functions are methods of Math. These include trigonometric, logarithmic, exponential, and other functions. For example, if you want to use the trigonometric function sine, you would write

```
Math.sin(1.56)
```

Note:
Trigonometric methods of Math take arguments in radians.

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)
}
```

JavaScript Math Object Properties

E	Returns Euler's constant (approx. 2.718)
LN2	Returns the natural logarithm of 2 (approx. 0.693)
LN10	Returns the natural logarithm of 10 (approx. 2.302)
LOG2E	Returns the base-2 logarithm of E (approx. 1.442)
LOG10E	Returns the base-10 logarithm of E (approx. 0.434)
PI	Returns PI (approx. 3.14159)
SQRT1_2	Returns the square root of 1/2 (approx. 0.707)
SQRT2	Returns the square root of 2 (approx. 1.414)

JavaScript Math Object Methods

abs(x)	Returns the absolute value of a number
acos(x)	Returns the arccosine of a number
asin(x)	Returns the arcsine of a number
atan(x)	Returns the arctangent of x as a numeric value between -PI/2 and PI/2 radians
atan2(y,x)	Returns the angle theta of an (x,y) point as a numeric value between -PI and PI radians
ceil(x)	Returns the value of a number rounded upwards to the nearest integer
cos(x)	Returns the cosine of a number
exp(x)	Returns the value of E^x
floor(x)	Returns the value of a number rounded downwards to the nearest integer
log(x)	Returns the natural logarithm (base E) of a number
max(x,y)	Returns the number with the highest value of x and y
min(x,y)	Returns the number with the lowest value of x and y
pow(x,y)	Returns the value of x to the power of y
random()	Returns a random number between 0 and 1
round(x)	Rounds a number to the nearest integer
sin(x)	Returns the sine of a number
sqrt(x)	Returns the square root of a number
tan(x)	Returns the tangent of an angle

JavaScript Number Object Description

The Number object has properties for numerical constants, such as maximum value, not-a-number, and infinity. You use these properties as follows:

```
biggestNum = Number.MAX_VALUE
smallestNum = Number.MIN_VALUE
infiniteNum = Number.POSITIVE_INFINITY
negInfiniteNum = Number.NEGATIVE_INFINITY
notANum = Number.NaN
```

JavaScript Number Object Properties

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

JavaScript String Object Description

JavaScript does not have a string data type. However, you can use the String object and its methods to work with strings in your applications. The String object has a large number of methods for manipulating strings. It has one property for determining the string's length.
To create a String object:

```
stringObjectName = new String(string)
```

Arguments:

- *stringObjectName* is the name of a new String object.
- *string* is any string.

For example, the following statement creates a String object called mystring:

JavaScript String Object Properties

length
Returns the number of characters in a string

JavaScript String Object Methods

anchor(*anchorname*)
Creates an HTML anchor

Argument:

- *anchorname* (Required) Defines a name for the anchor

In this example we will add an anchor to a text:

```
var txt="Hello world!";
document.write(txt.anchor("myanchor"))
```

The code above could be written in plain HTML, like this:

```
<a name="myanchor">Hello world!</a>
```

The output of the code above will be:

Hello world!

big()
Displays a string in a big font

In this example "Hello world!" will be displayed in a big font:

The output of the code above will be:

For example, the following statement creates a String object called mystring:

```
mystring = new String ("Hello, World!")
```

String literals are also String objects; for example, the literal "Howdy" is a String object.

A String object has one property, length, that indicates the number of characters in the string. So, using the previous example, the expression

```
x = mystring.length
```

assigns a value of 13 to x, because "Hello, World!" has 13 characters.

A String object has two types of methods: those that return a variation on the string itself, such as substring and toUpperCase, and those that return an HTML-formatted version of the string, such as bold and link.

For example, using the previous example, both mystring.toUpperCase() and "hello, world!".toUpperCase() return the string "HELLO, WORLD!".

The substring method takes two arguments and returns a subset of the string between the two arguments. Using the previous example, mystring.substring(4, 9) returns the string "o, Wo." For more information, see the reference topic for substring.

The String object also has a number of methods for automatic HTML formatting, such as bold to create boldface text and link to create a hyperlink. For example, you could create a hyperlink to a hypothetical URL with the link method as follows:

```
mystring.link("http://www.helloworld.com")
```

		big font:
		`var str="Hello world!";`
		`document.write(str.big());`
			Hello world!

blink()	Displays a blinking string **Note:** This method does not work in Internet Explorer.	`var str="Hello world!";` `document.write(str.blink());`	The output of the code above will be: *Hello world!*

bold()	Displays a string in bold	In this example "Hello world!" will be displayed in bold:
		`var str="Hello world!";`
		`document.write(str.bold())`
			The output of the code above will be:
			Hello world!

charAt(*index*)	Returns the character at a specified position
	Argument:
	• *index* (Required) A number representing a position in the string
	Note: The first character in the string is at position 0.
		In the string "Hello world!", we will return the character at position 1:
		`var str="Hello world!";`
		`document.write(str.charAt(1))`
			The output of the code above will be:
			e

charCodeAt(*index*)	Returns the Unicode of the character at a specified position
	Argument:
	• *index* (Required) A number representing a position in the string
	Note: The first character in the string is at position 0.
		In the string "Hello world!", we will return the Unicode of the character at position 1:
		`var str="Hello world!";`
		`document.write(str.charCodeAt(1))`
			The output of the code above will be:
			101

concat(*stringX*,*stringX*,...,*stringX*)	Joins two or more strings
	Argument:
	• *stringX* (Required) One or more string objects to be joined to a string
		In the following example we will create two strings and show them as one using concat():
		`var str1="Hello ";`
		`var str2="world!";`
		`document.write(str1.concat(str2));`
			The output of the code above will be:
			Hello world!

fixed()	Displays a string as teletype text	In this example "Hello world!" will be displayed as teletype text:
		`var str="Hello world!";`
		`document.write(str.fixed())`
			The output of the code above will be:
			`Hello world!`

fontcolor(*color*)	Displays a string in a specified color
	Argument:
	• *color*(Required) Specifies a font-color for the string. The value can be a color name (red), an RGB value (rgb(255,0,0)), or a hex number (#FF0000)
		In this example "Hello world!" will be displayed in red:
		`var str="Hello world!";`
		`document.write(str.fontcolor("Red"))`
			The output of the code above will be:
			Hello world!

fontsize(*size*)	Displays a string in a specified size
	Argument:
	• size(Required) A number that specifies the font size
	Note: The size parameter must be a number from 1 to 7.
		In this example "Hello world!" will be displayed in a large font-size:
		`var str="Hello world!";`
		`document.write(str.fontsize(7))`
			The output of the code above will be:
			# *Hello world!*

fromCharCode(*numX*,*numX*,...,*numX*)	Takes the specified Unicode values and returns a string **Argument:** • *numX*(Required) One or more Unicode values **Note:** This method is a static method of String - it is not used as a method of a String object that you have created. The syntax is always String.fromCharCode() and not myStringObject.fromCharCode().	In this example we will write "HELLO" and "ABC" from Unicode: `document.write(String.fromCharCode(72,69,76,76,79));` `document.write(" ");` `document.write(String.fromCharCode(65,66,67))`	The output of the code above will be: *HELLO* *ABC*
indexOf(*searchvalue*,*fromindex*)	Returns the position of the first occurrence of a specified string value in a string **Arguments:** • *searchvalue*(Required) Specifies a string value to search for • *fromindex*(Optional) Specifies where to start the search **Notes:** • The indexOf() method is case sensitive! • This method returns -1 if the string value to search for never occurs.	In this example we will do different searches within a "Hello world!" string: `var str="Hello world!";` `document.write(str.indexOf("Hello") + " ");` `document.write(str.indexOf("World") + " ");` `document.write(str.indexOf("world"));`	The output of the code above will be: *0* *-1* *6*

italics()	Displays a string in italic	In this example "Hello world!" will be displayed in italic:
		`var str="Hello world!";`
		`document.write(str.italics())`
			The output of the code above will be:
			Hello world!

lastIndexOf(*searchvalue*,*fromindex*)	Returns the position of the last occurrence of a specified string value, searching backwards from the specified position in a string **Arguments:** • *search*(Required) Specifies a string value to search for • *fromindex*(Optional) Specifies where to start the search. Starting backwards in the string **Notes:** • The indexOf() method is case sensitive! • This method returns -1 if the string value to search for never occurs.	In this example we will do different searches within a "Hello world!" string: `var str="Hello world!";` `document.write(str.lastIndexOf("Hello") + " ");` `document.write(str.lastIndexOf("World") + " ");` `document.write(str.lastIndexOf("world"))`	The output of the code above will be: *0* *-1* *6*

link()	Displays a string as a hyperlink	In this example "Free Web Tutorials!" will be displayed as a hyperlink:
		`var str="Free Web Manuals!";`
		`document.write(str.link("http://www.cheat-sheets.org/"))`
			The output of the code above will be:
			Free Web Manuals!

match(*searchvalue*)	Searches for a specified string value in a string This method is similar to indexOf() and lastIndexOf(), but it returns the specified string, instead of the position of the string. **Argument:** • *searchvalue*(Required) Specifies a string value to search for **Notes:** • The match() method is case sensitive! • This method returns null if the string value to search for never occurs.	`var str="Hello world!";` `document.write(str.match("world") + " ");` `document.write(str.match("World") + " ");` `document.write(str.match("worlld") + " ");` `document.write(str.match("world!"))`	The output of the code above will be: *world* *null* *null* *world!*

replace(*findstring*,*newstring*)	Replaces some characters with some other characters in a string
	Arguments:
	• *findstring*(Required) Required. Specifies a string value to find. To perform a global search add a 'g' flag to this parameter and to perform a case-insensitive search add an 'i' flag
	• *newstring*(Required) Specifies the string to replace the found value from findstring
		In the following example we will replace the word Microsoft with MANUALS.SU:
		`var str="Visit Microsoft!";`
		`document.write(str.replace(/Microsoft/, "MANUALS.SU"))`
			The output of the code above will be:
			Visit MANUALS.SU!

Note:

- The replace() method is case sensitive.

search(*searchstring*)	Searches a string for a specified value **Argument:** • *searchstring*(Required) Required. The value to search for in a string. To perform a case-insensitive search add an 'i' flag **Notes:** • The search() method is case sensitive. • The search() method returns the position of the specified value in the string. If no match was found it returns -1.	In the following example we will search for the word "MANUALS.SU": *var str="Visit MANUALS.SU!";* *document.write(str.search(/MANUALS.SU/))*	The output of the code above will be: *6*
slice(*start*,*end*)	Extracts a part of a string and returns the extracted part in a new string **Argument:** • *start*(Required) Specify where to start the selection. Must be a number • *end*(Optional) Specify where to end the selection. Must be a number **Notes:** • You can use negative index numbers to select from the end of the string. • If end is not specified, slice() selects all characters from the specified start position and to the end of the string.	In this example we will extract all characters from a string, starting at position 6: *var str="Hello happy world!";* *document.write(str.slice(6))*	The output of the code above will be: *happy world!*
small()	Displays a string in a small font	In this example "Hello world!" will be displayed in a small font: *var str="Hello world!";* *document.write(str.small())*	The output of the code above will be: *Hello world!*
split(*separator, howmany*)	Splits a string into an array of strings **Arguments:** • *separator*(Required) Specifies the character, regular expression, or substring that is used to determine where to split the string • *howmany*(Optional) Specify how many times split should occur. Must be a numeric value **Note:** • If an empty string ("") is used as the separator, the string is split between each character.	In this example we will split up a string in different ways: *var str="How are you doing today?";* *document.write(str.split(" ") + " ");* *document.write(str.split("") + " ");* *document.write(str.split(" ",3))*	The output of the code above will be: *How,are,you,doing,today?* *H,o,w, ,a,r,e, ,y,o,u, ,d,o,i,n,g, ,t,o,d,a,y,?* *How,are,you*
strike()	Displays a string with a strikethrough	In this example "Hello world!" will be displayed with a line trough it: *var str="Hello world!";* *document.write(str.strike())*	The output of the code above will be: ~~*Hello world!*~~
sub()	Displays a string as subscript	In this example "Hello world!" will be displayed in subscript: *var str="Hello world!";* *document.write(str.sub())*	The output of the code above will be: *Hello world!*
substr(*start*,*length*)	Extracts a specified number of characters in a string, from a start index **Arguments:** • *start*(Required) Where to start the extraction. Must be a numeric value • *length*(Optional) How many characters to extract. Must be a numeric value. **Notes:** • To extract characters from the end of the string, use a negative start number. • The start index starts at 0. • If the length parameter is omitted, this method extracts to the end of the string.	In this example we will use substr() to extract some characters from a string: *var str="Hello world!";* *document.write(str.substr(3))*	The output of the code above will be: *lo world!*
substring(*start*,*stop*)	Extracts the characters in a string between two specified indices **Arguments:** • *start*(Required) Where to start the extraction. Must be a numeric value • *stop*(Optional) Where to stop the extraction. Must be a numeric value **Notes:** • To extract characters from the end of the string, use a negative start number. • The start index starts at 0. • If the stop parameter is omitted, this method extracts to the end of the string.	In this example we will use substring() to extract some characters from a string: *var str="Hello world!";* *document.write(str.substring(3))*	The output of the code above will be: *lo world!*
sup()	Displays a string as superscript	In this example "Hello world!" will be displayed in superscript: *var str="Hello world!";* *document.write(str.sup())*	The output of the code above will be: *Hello world!*
toLowerCase()	Displays a string in lowercase letters	In this example "Hello world!" will be displayed in lower case letters: *var str="Hello World!";* *document.write(str.toLowerCase())*	The output of the code above will be: *hello world!*
toUpperCase()	Displays a string in uppercase letters	In this example "Hello world!" will be displayed in upper case letters: *var str="Hello world!";* *document.write(str.toUpperCase())*	The output of the code above will be: *HELLO WORLD!*

JavaScript Event

onabort	Loading of an image is interrupted
onblur	An element loses focus
onchange	The content of a field changes
onclick	Mouse clicks an object
ondblclick	Mouse double-clicks an object
onerror	An error occurs when loading a document or an image
onfocus	An element gets focus
onkeydown	A keyboard key is pressed
onkeypress	A keyboard key is pressed or held down
onkeyup	A keyboard key is released
onload	A page or an image is finished loading
onmousedown	A mouse button is pressed
onmousemove	The mouse is moved
onmouseout	The mouse is moved off an element
onmouseover	The mouse is moved over an element
onmouseup	A mouse button is released
onreset	The reset button is clicked
onresize	A window or frame is resized
onselect	Text is selected

Other

onsubmit	The submit button is clicked
onunload	The user exits the page