JavaScript syntax

This article is part of the JavaScript series.
JavaScript
JavaScript syntax
JavaScript topics

The syntax of JavaScript is a set of rules that defines what constitutes a valid program in the JavaScript language.

ⓘ **Note:** The examples below often use an alert function for standard text output. The JavaScript standard library lacks an official standard text output function. However, given that JavaScript is mainly used as a client-side scripting language within modern web browsers, and that almost all web browsers provide access to the alert function, alert is included in our examples.

Origin of Syntax

Brendan Eich summarized the ancestry of the syntax in the first paragraph of the JavaScript 1.1 [1] specification as follows:

> JavaScript borrows most of its syntax from Java, but also inherits from Awk and Perl, with some indirect influence from Self in its object prototype system.

Syntax Basics

Case sensitivity

JavaScript is case sensitive. It is common to start constructor names with a capitalised letter and functions or variables with a lower-case letter.

Whitespace and semicolons

Spaces, tabs and newlines used outside of string constants are called whitespace. Unlike C, whitespace in JavaScript source can directly impact semantics. Because of a technique called "semicolon insertion", some statements that are well formed when a newline is parsed will be considered complete (as if a semicolon were inserted just prior to the newline). Programmers are advised to supply statement-terminating semicolons explicitly, although it degrades readability, because it may lessen unintended effects of the automatic semicolon insertion.[2]

```
return
a + b;

// Returns undefined. Treated as:
//   return;
//   a + b;
```

But:

```
a = b +
c(d + e).foo()

// Treated as:
//   a = b + c(d + e).foo();
```

Comments

Comment syntax is the same as in C++ and many other languages.

```
// a short, one-line comment

/* this is a long, multi-line comment
   about my script. May it one day
   be great. */
```

Note that JavaScript explicitly forbids nesting of comments, e.g.

```
/* You can't do
   /* this */
*/
/* But you can
//Do this.
*/

// And you can /* also do this */


// As /* well /* as this */ */
```

Variables

Variables in standard JavaScript have no type attached, and any value can be stored in any variable. Variables can be declared with a var statement. These variables are lexically scoped and once a variable is declared, it may be accessed anywhere inside the function where it is declared. Variables declared outside any function, and variables first used within functions without being declared with 'var', are global. Here is an example of variable declarations and global values:

```
x = 0; // A global variable
var y = 'Hello!'; // Another global variable

function f(){
  var z = 'foxes'; // A local variable
  twenty = 20; // Global because keyword var is not used
  return x; // We can use x here because it is global
}
// The value of z is no longer available
```

Primitive data types

The JavaScript language provides a handful of primitive data types. Some of the primitive data types also provide a set of named values that represent the extents of the type boundaries. These named values are described within the appropriate sections below.

Undefined

The value of undefined is assigned to all uninitialized variables, and is also returned when checking for object properties that do not exist. In a Boolean context, the undefined value is considered a false value in JavaScript.

Note: Undefined is a true primitive-type within the JavaScript language. As such, when performing checks that enforce type checking, the undefined value will not equal other false types.

```
var test;                    // variable declared but not defined, ...
                             // ... set to value of undefined
var testObj = {};
alert(test);                 // test variable exists but value not ...
                             // ... defined, displays undefined
alert(testObj.myProp);       // testObj exists, property does not, ...
                             // ... displays undefined
alert(undefined == null);    // unenforced type during check, displays
true
alert(undefined === null); // enforce type during check, displays false
```

Note: There is no built-in language literal for undefined. Thus `(x == undefined)` is not a foolproof way to check whether a variable is undefined, because it is legal for someone to write `var undefined = "I'm defined now";`. A more robust comparison can be made using `(typeof x == 'undefined')` or a function like this:

```
function isUndefined(x) { var u; return x === u; }
```

or

```
function isUndefined(x) { return x === void(0); }
```

Null

Unlike undefined, null is often set to indicate that something has been declared but has been defined to be empty. In a Boolean context, the value of null is considered a false value in JavaScript.

Note: Null is a true primitive-type within the JavaScript language, of which `null` (note case) is the single value. As such, when performing checks that enforce type checking, the null value will not equal other false types.

```
alert(null == undefined);  // unenforced type during check, displays
true
alert(null === undefined); // enforce type during check, displays false
```

Number

Numbers in JavaScript are represented in binary as IEEE-754 Doubles, which provides an accuracy to about 14 or 15 significant digits JavaScript FAQ, Numbers [3]. Because they are floating point numbers, they do not always exactly represent real numbers, including fractions.

This becomes an issue when formatting numbers. For example:

```
alert(0.94 - 0.01); // displays 0.9299999999999999
```

As a result, a routine such as the toFixed() method should be used to round numbers whenever they are formatted for output [4].

Numbers may be specified in any of these notations:

```
345;    // an "integer", although there is only one numeric type in
JavaScript
34.5;   // a floating-point number
3.45e2; // another floating-point, equivalent to 345
0377;   // an octal integer equal to 255
0xFF;   // a hexadecimal integer equal to 255, digits represented by
the ...
        // ... letters A-F may be upper or lowercase
```

In some ECMAScript implementations such as ActionScript, RGB color values are sometimes specified with hexadecimal integers:

```
var colorful = new Color( '_root.shapes' );
colorful.setRGB( 0x003366 );
```

The extents of the number type may also be described by named constant values:

```
Infinity;  // Construct equivalent to positive Infinity
-Infinity; // Negated Infinity construct, equal to negative Infinity
NaN;       // The Not-A-Number value, often returned as a failure in ...
           // ... string-to-number conversions
```

The Number constructor, or a unary + or -, may be used to perform explicit numeric conversion:

```
var myString = "123.456";
var myNumber1 = Number( myString );
var myNumber2 = + myString;
```

When used as a constructor, a numeric *wrapper* object is created, (though it is of little use):

```
myNumericWrapper = new Number( 123.456 );
```

String

Strings in Javascript are a sequence of characters. Strings in JavaScript can be created directly by placing the series of characters between double or single quotes.

```
var greeting = "Hello, world!";
var another_greeting = 'Greetings, people of Earth.';
```

Strings are instances of the String class:

```
var greeting = new String("Hello, world!");
```

You can access individual characters within a string using the charAt() method (provided by String.prototype). This is the preferred way when accessing individual characters within a string, as it also works in non-Mozilla-based browsers:

```
var h = greeting.charAt(0); // Now h contains 'H' - Works in both ...
                            // ... Internet Explorer and Mozilla ...
                            // ... based browsers
```

In Mozilla based browsers, individual characters within a string can be accessed (as strings with only a single character) through the same notation as arrays:

```
var h = greeting[0]; // Now h contains 'H' - Works in Mozilla based
browsers
```

But JavaScript strings are immutable:

```
greeting[0] = "H"; // ERROR
```

Applying the equality operator ("==") to two strings returns true if the strings have the same contents, which means: of same length and same cases (for alphabets). Thus:

```
var x = "world";
var compare1 = ("Hello, " + x == "Hello, world"); // Now compare1
contains true
var compare2 = ("Hello, " + x == "hello, world"); // Now compare2
contains  ...
                                                  // ... false since
the ...
                                                  // ... first
characters ...
                                                  // ... of both
operands ...
                                                  // ... are not of the
 same case
```

You cannot use quotes of the same type inside the quotes unless they are escaped.

```
var x = '"Hello, world!" he said.' //Just fine.
var x = ""Hello, world!" he said." //Not good.
var x = "\"Hello, world!\" he said." //That works by replacing " with \"
```

Boolean

JavaScript does not have an explicit Boolean data type but has two Boolean values represented by the keywords true and false. These values are often returned by JavaScript logic operations. However many values will evaluate to false when used in a logical context, including zero, null, zero length strings, and unknown properties of objects. All other variable values, including empty arrays and empty objects, will evaluate to true. Conversely, the values true and false can map to a corresponding number value in arithmetic contexts.

```
// If we do not enforce type checking...
alert(true == 1);     // is True -- true is logically equal to 1, ...
                      // ... as they are both truthy values
alert(false == 0);    // is True -- false is logically equal to 0, ...
                      // ... as they are both falsey values
// If we enforce type checking...
alert(true === 1);    // is False -- true is not strictly equal to 1,
...
                      // ... as they are different types
alert(false === 0);   // is False -- false is not strictly equal to 0,
...
                      // ... as they are different types
```

The language does offer a Boolean object which can be used as a wrapper for handling Boolean values. However, a Boolean object will always evaluate to true even if it has a value of false.

```
var objBool = new Boolean(false);

if ( false || 0 || "" || null || window.not_a_property ) {
    alert("never this");
} else if ( true && [] && {} && objBool ) {
    alert("Hello Wikipedia");              // will bring up this message
}
```

Native Objects

The JavaScript language provides a handful of native objects. JavaScript native objects are considered part of the JavaScript specification. JavaScript environment notwithstanding, this set of objects should always be available.

Array

An Array is a native JavaScript type specifically designed to store data values indexed by integer keys. Arrays, unlike the basic Object type, are prototyped with methods and properties to aid the programmer in routine tasks (e.g., join, slice, and push).

As in the C family, arrays use a zero-based indexing scheme: A value that is inserted into an empty array by means of the push method occupies the 0th index of the array.

```
var myArray = [];             // Point the variable myArray to a newly
...
                              // ... created, empty Array
myArray.push("hello world"); // Fill the next empty index, in this case
 0
alert(myArray[0]);            // Equivalent to alert("hello world");
```

Arrays have a length property that is guaranteed to always be larger than the largest integer index used in the array. It is automatically updated if one creates a property with an even larger index. Writing a smaller number to the length property will remove larger indices.

Elements of Arrays may be accessed using normal object property access notation:

```
myArray[1];   //this gives you the 2nd item in myArray
myArray["1"];
```

The above two are equivalent. It's not possible to use the "dot"-notation or strings with alternative representations of the number:

```
myArray.1;     // syntax error
myArray["01"]; // not the same as myArray[1]
```

Declaration of an array can use either an Array literal or the Array constructor:

```
myArray = [0,1,,,4,5];            // array with length 6 and 6
elements, ...
                                  // ... including 2 undefined elements
myArray = new Array(0,1,2,3,4,5); // array with length 6 and 6 elements
myArray = new Array(365);         // an empty array with length 365
```

Arrays are implemented so that only the elements defined use memory; they are "sparse arrays". Setting myArray[10] = 'someThing' and myArray[57] = 'somethingOther' only uses space for these two elements, just like any other object. The length of the array will still be reported as 58.

You can use the object declaration literal to create objects that behave much like associative arrays in other languages:

```
dog = {"color":"brown", "size":"large"};
dog["color"]; // this gives you "brown"
dog.color;    // this also gives you "brown"
```

You can use the object and array declaration literals to quickly create arrays that are associative, multidimensional, or both.

```
cats = [{"color":"brown", "size":"large"},
        {"color":"black", "size":"small"}];
cats[0]["size"];      // this gives you "large"

dogs = {"rover":{"color":"brown", "size":"large"},
        "spot":{"color":"black", "size":"small"}};
dogs["spot"]["size"]; // this gives you "small"
dogs.rover.color;     // this gives you "brown"
```

Date

A Date object stores a signed millisecond count with zero representing 1970-01-01 00:00:00 UT and a range of $\pm 10^8$ days. One can create a Date object representing "now":

```
var d = new Date();
```

Initial value of Date object is current date and time. Other ways to construct a Date instance:

```
var d = new Date(2010,4,5);           // create a new Date instance with
 value:
                                      //     2010-4-5 00:00:00
var d = new Date(2010,4,6,14,25,30); // create a new Date instance with
 value:
                                      //     2010-4-5 14:25:30
```

Format the display of a Date instance:

```
var d = new Date(2010,4,6,14,25,30); // May,2010
var display = d.getFullYear() + '-' + (d.getMonth()+1) + '-' +
d.getDate()
            + ' ' + d.getHours() + ':' + d.getMinutes() + ':' +
d.getSeconds();
alert(display);                       // Displays '2010-5-6 14:25:30'
```

Construct from a String:

```
var d = new Date("2010-4-6 14:25:30");
alert(d);
```

Error

Custom error messages can be created using the Error class:

```
throw new Error("What the...");
```

Nested within conditional statements, such instantiations can substitute for try/catch blocks:

```
var eMail = prompt("Please enter your e-mail address:", "");
if (!eMail || eMail.length == 0)
{
      throw new Error("Excuse me: You must enter your e-mail address to
 continue.");
}
```

Math

The Math object contains various math-related constants (e.g. π) and functions (e.g. cosine). (Note that it is not a constructor like Array or Date.) All the trigonometric functions use angles expressed in radians; not degrees or grads.

Properties of the Math object

Property	Value Returned rounded to 5 digits	Description
Math.E	2.7183	*e*: Euler's number
Math.LN2	0.69315	Natural logarithm of 2
Math.LN10	2.3026	Natural logarithm of 10
Math.LOG2E	1.4427	Logarithm to the base 2 of *e*
Math.LOG10E	0.43429	Logarithm to the base 10 of *e*
Math.PI	3.14159	π: circumference/diameter of a circle
Math.SQRT1_2	0.70711	Square root of ½
Math.SQRT2	1.4142	Square root of 2

Methods of the Math object

Example	Value Returned rounded to 5 digits	Description
Math.abs(-2.3)	2.3	Absolute value: (x < 0) ? -x : x
Math.acos(Math.SQRT1_2)	0.78540 rad. = 45°	Arccosine
Math.asin(Math.SQRT1_2)	0.78540 rad. = 45°	Arcsine
Math.atan(1)	0.78540 rad. = 45°	Half circle arctangent (-π/2 to +π/2)
Math.atan2(-3.7, -3.7)	-2.3562 rad. = -135°	Whole circle arctangent (-π to +π)
Math.ceil(1.1)	2	Ceiling: round up to smallest integer ≥ argument
Math.cos(Math.PI/4)	0.70711	Cosine
Math.exp(1)	2.7183	Exponential function: *e* raised to this power
Math.floor(1.9)	1	Floor: round down to largest integer ≤ argument
Math.log(Math.E)	1	Natural logarithm, base *e*
Math.max(1, -2)	1	Maximum: (x > y) ? x : y
Math.min(1, -2)	-2	Minimum: (x < y) ? x : y
Math.pow(-3, 2)	9	Exponentiation (raised to the power of): Math.pow(x, y) gives x^y
Math.random()	0.17068	Pseudorandom number between 0 (inclusive) and 1 (exclusive)
Math.round(1.5)	2	Round to the nearest integer; half fractions are rounded up (e.g. 1.5 rounds to 2)
Math.sin(Math.PI/4)	0.70711	Sine
Math.sqrt(49)	7	Square root
Math.tan(Math.PI/4)	1	Tangent

Regular Expression

```
/expression/.test(string);
"string".search(/expression/);
"string".replace(/expression/,replacement);

// Here are some examples
if(/Tom/.test("My name is Tom")) alert("Hello Tom!");
alert("My name is Tom".search(/Tom/));                    // == 11
(letters before Tom)
alert("My name is Tom".replace(/Tom/,"John"));            // == "My
name is John"
```

Character Classes:

```
// \d   - digit
// \D   - non digit
// \s   - space
// \S   - non space
// \w   - word char
// \W   - non word
// [ ]  - one of
// [^ ] - one not of
//  -   - range

if(/\d/.test('0'))                    alert('Digit');
if(/[0-9]/.test('5'))                 alert('Digit');
if(/[13579]/.test('1'))               alert('Odd number');
if(/\S\S\s\S\S\S\S/.test('My name')) alert('Format OK');
if(/\w\w\w/.test('Tom'))              alert('Format OK');
if(/[a-z]/.test('b'))                 alert('Small letter');
if(/[A-Z]/.test('B'))                 alert('Big letter');
if(/[a-zA-Z]/.test('B'))              alert('Letter');
```

Character matching:

```
// A...Z a...z 0...9  - alphanumeric
// \u0000...\uFFFF    - Unicode hexadecimal
// \x00...\xFF        - ASCII hexadecimal
// \t                 - tab
// \n                 - new line
// \r                 - CR
// .                  - any character
// |                  - OR

if(/T.m/.test('Tom')) alert ('Hi Tom, Tam or Tim');
if(/A|B/.test("A"))   alert ('A or B');
```

Repeaters:

```
// ?     - 0 or 1 match
// *     - 0 or more
// +     - 1 or more
// {n}   - exactly n
// {n,}  - n or more
// {0,n}  - n or less
// {n,m} - range n to m

if(/ab?c/.test("ac"))        alert("OK"); // match: "ac", "abc"
if(/ab*c/.test("ac"))        alert("OK"); // match: "ac", "abc",
"abbc", "abbbc" etc.
if(/ab+c/.test("abc"))       alert("OK"); // match: "abc", "abbc",
"abbbc" etc.
if(/ab{3}c/.test("abbbc"))   alert("OK"); // match: "abbbc"
if(/ab{3,}c/.test("abbbc"))  alert("OK"); // match: "abbbc", "abbbbc",
"abbbbbc" etc.
if(/ab{1,3}c/.test("abc"))   alert("OK"); // match: "abc","abbc",
"abbbc"
```

Anchors:

```
// ^   - string starts with
// $   - string ends with

if(/^My/.test("My name is Tom"))   alert ("Hi!");
if(/Tom$/.test("My name is Tom"))  alert ("Hi Tom!");
```

Subexpression

```
// ( )   - groups characters

if(/water(mark)?/.test("watermark"))  alert("Here is water!"); //
match: "water", "watermark",
if(/(Tom)|(John)/.test("John"))       alert("Hi Tom or John!");
```

Flags:

```
// /g   - global
// /i   - ignore upper/lower case
// /m   -

alert("hi tom!".replace(/Tom/i,"John"));      // == "hi John!"
alert("ratatam".replace(/ta/,"tu"));          // == "ratutam"
alert("ratatam".replace(/ta/g,"tu"));         // == "ratutum"
```

Advanced methods

```
my_array=my_string.split(my_delimiter);
// example
my_array="dog,cat,cow".split(",");            //
my_array==("dog","cat","cow");
```

```
my_array=my_string.match(my_expression);
// example
my_array="We start at 11:30, 12:15 and 16:45".match(/\d\d:\d\d/); //
my_array=("11:30","12:15","16:45");
```

Capturing groups

```
var myRe = /(\d{4}-\d{2}-\d{2}) (\d{2}:\d{2}:\d{2})/;
var results = myRe.exec("The date and time are 2009-09-08 09:37:08.");
if(results) {
    alert("Matched: " + results[0]); // Entire match
    var my_date = results[1]; // First group == "2009-09-08"
    var my_time = results[2]; // Second group == "09:37:08"
    alert("It is " + my_time + " on " + my_date);
} else alert("Did not find a valid date!");
```

Function

Every function in javascript is an instance of the Function object:

```
var add=new Function('x','y','return x+y');//x,y is the
argument.'return x+y' is the function body, which is the last in the
argument list.
var t=add(1,2);
alert(t);//3
```

The add function above is often defined as below:

```
function add(x,y)
{
  return x+y;
}
var t=add(1,2);
alert(t);//3
```

The later is better,because it is more intuitive,and faster. A function instance has properties and methods.

```
function subtract(x,y)
{
  return x-y;
}
alert(subtract.length);//2,expected amount of arguments.
alert(subtract.toString());
/*
function subtract(x,y)
{
  return x-y;
}
*/
```

Operators

The '+' operator is overloaded; it is used for string concatenation and arithmetic addition and also to convert strings to numbers. It also has special meaning when used in a regular expression.

```
// Concatenate 2 strings
var a = 'This';
var b = ' and that';
alert(a + b);  // displays 'This and that'

// Add two numbers
var x = 2;
var y = 6;
alert(x + y); // displays 8

// Adding a string and a number results in concatenation
alert(x + '2'); // displays 22

// Convert a string to a number
var z = '4';   // z is a string (the digit 4)
var x = '2';   // x is a string (the digit 2)
alert(z + x); // displays 42
alert(+z + +x);// displays 6
```

Arithmetic

Binary operators

```
+     Addition
-     Subtraction
*     Multiplication
/     Division (returns a floating-point value)
%     Modulus (returns the integer remainder)
```

Unary operators

```
+     Unary conversion of string to number
-     Unary negation (reverses the sign)
++    Increment (can be prefix or postfix)
--    Decrement (can be prefix or postfix)
```

```
var x = 1;
alert( ++x );  // displays: 2
alert( x++ );  // displays: 2
alert( x );    // displays: 3
alert( x-- ); //  displays: 3
alert( x );   //  displays: 2
alert( --x ); //  displays: 1
```

Assignment

```
=     Assign
+=    Add and assign
-=    Subtract and assign
*=    Multiply and assign
/=    Divide and assign
%=    Modulus and assign
```

```
var x = 1;
x *= 3;
alert( x );  // displays: 3
x /= 3;
alert( x );  // displays: 1
x -= 1;
alert( x );  // displays: 0
```

Comparison

```
==    Equal
!=    Not equal
>     Greater than
>=    Greater than or equal to
<     Less than
<=    Less than or equal to
===   Identical (equal and of the same type)
!==   Not identical
```

Boolean

JavaScript has three logical boolean operators: *&&* (logical AND), || (logical OR), and *!* (logical NOT).

In the context of a boolean operation, all JavaScript values evaluate to true unless the value is the boolean *false* itself, the number 0, a string of length 0 (empty string), or one of the special values *null*, *undefined*, or *NaN*. The Boolean function can be used to explicitly perform this conversion:

```
Boolean( false );     // returns false
Boolean( 0 );         // returns false
Boolean( 0.0 );       // returns false
Boolean( "" );        // returns false
Boolean( null );      // returns false
Boolean( undefined ); // returns false
Boolean( NaN );       // returns false
// ONLY empty strings return false
Boolean("false");     // returns true
Boolean("0");         // returns true
```

The unary NOT operator *!* first evaluates its operand in a boolean context, and then returns the opposite boolean value:

```
var a = 0;
var b = 9;
```

```
!a; // evaluates to true,  just as (Boolean( a ) == false)
!b; // evaluates to false, just as (Boolean( b ) == true)
```

A double use of the *!* operator can be used to *normalize* a boolean value:

```
var arg = null;
arg = !!arg; // arg is now the value false, rather than null

arg = "finished"; // non-empty string
arg = !!arg; // arg is now the value true
```

In the earliest implementations of JavaScript and JScript, the *&&* and || operators behaved in the same manner as their counterparts in other C derived programming languages, in that they always returned a boolean value:

```
x && y; // returns true if x AND y evaluate to true: (Boolean( x ) ==
Boolean( y ) == true), false otherwise
x || y; // returns true if x OR y evaluates to true, false otherwise
```

In the newer implementations, these operators return one of their operands:

```
expr1 && expr2; // returns expr1 if it evaluates to false, otherwise it
 returns expr2
expr1 || expr2; // returns expr1 if it evaluates to true, otherwise it
returns expr2
```

This novel behavior is little known even among experienced JavaScripters, and can cause problems if one expects an actual boolean value.

- Short-circuit logical operations means the expression will be evaluated from left to right until the answer can be determined. For example:

a || b is automatically true if a is true. There is no reason to evaluate b. a && b is false if a is false. There is no reason to evaluate b.

```
&&    and
||    or
!     not (logical negation)
```

Bitwise

Binary operators

```
&     And
|     Or
^     Xor

<<    Shift left  (zero fill)
>>    Shift right (sign-propagating); copies of the leftmost bit
(sign bit) are shifted in from the
      left.
>>>   Shift right (zero fill)

      For positive numbers, >> and >>> yield the same
result.
```

Unary operators

```
~      Not (inverts the bits)
```

String

```
=      Assignment
+      Concatenation
+=     Concatenate and assign
```

Examples

```
str = "ab" + "cd";    // "abcd"
str += "e";           // "abcde"

str2 = "2"+2          // "22", not "4" or 4.
```

Control structures

Compound statements

A pair of curly brackets `{ }` and an enclosed sequence of statements constitute a compound statement, which can be used wherever a statement can be used.

If ... else

```
if (expr)
{
  //statements;
}
else if (expr2)
{
  //statements;
}
else
{
  //statements;
}
```

Conditional operator

The conditional operator creates an expression that evaluates as one of two expressions depending on a condition. This is similar to the *if* statement that selects one of two statements to execute depending on a condition. I.e., the conditional operator is to expressions what *if* is to statements.

```
var result = (condition) ? expression : alternative;
```

is the same as:

```
if (condition)
{
  result = expression;
}
```

```
else
{
  result = alternative;
}
```

Unlike the *if* statement, the conditional operator cannot omit its "else-branch".

Switch statement

```
switch (expr) {
  case SOMEVALUE:
    //statements;
    break;
  case ANOTHERVALUE:
    //statements;
    break;
  default:
    //statements;
    break;
}
```

- break; is optional; however, it is usually needed, since otherwise code execution will continue to the body of the next case block.
- Add a break statement to the end of the last case as a precautionary measure, in case additional cases are added later.
- Strings can be used for the case values.
- Braces are required.

For loop

```
for (initial;condition;loop statement) {
  /*
    statements will be executed every time
    the for{} loop cycles, while the
    condition is satisfied
  */
}
```

or

```
for (initial;condition;loop statement) // one statement
```

For ... in loop

```
for (var property-name in object-name) {
  //statements using object-name[property-name];
}
```

- Iterates through all enumerable properties of an object.
- *Sources differ on whether this is usable for arrays*[5] .
- There are differences between the various web browsers with regard to which properties will be reflected with the for...in loop statement. In theory, this is controlled by an internal state property defined by the ECMAscript standard called "DontEnum", but in practice each browser returns a slightly different set of properties during introspection.

While loop

```
while (condition) {
  statement;
  statement;
  statement;
  ...
}
```

Do ... while

```
do {
  statement;
  statement;
  statement;
  ...
} while (condition);
```

With

The with statement sets the default object for the set of statements that follow.

```
with(document) {
  var a = getElementById('a');
  var b = getElementById('b');
  var c = getElementById('c');
};
```

- Note the absence of document. before each getElementById() invocation.

The semantics are similar to the with statement of Pascal.

Functions

A function is a block with a (possibly empty) parameter list that is normally given a name. A function may utilize local variables. If exit is not by a return statement, the value undefined is returned.

```
function gcd(segmentA, segmentB) {
  var diff = segmentA - segmentB
  if (diff == 0) return segmentA
  if (diff > 0)
     return gcd(segmentB, diff)
  else
     return gcd(segmentA, -diff)
}
alert(gcd(60, 40)); // 20
```

The number of arguments given when calling a function may not necessarily correspond to the number of arguments in the function definition; a named argument in the definition that does not have a matching argument in the call will have the value undefined. Within the function, the arguments may also be accessed through the arguments object; this provides access to all arguments using indices (e.g. arguments[0], arguments[1], ... arguments[n]), including those beyond the number of named arguments. Note that while the arguments list has a .length property, it is *not* an instance of Array; it does not have methods such as .slice(), .sort(), etc.

All parameters are passed by value (for objects it is the reference to the object that is passed).

```
var obj1 = {a:1}
var obj2 = {b:2}
function foo(p) {
  p = obj2; // ignores actual parameter
  p.b = arguments[1]
}
foo(obj1, 3) // Does not affect obj1 at all.  3 is additional parameter
alert(obj1.a + " " + obj2.b); // writes 1 3
```

Functions can be declared inside other functions, and access the outer function's local variables. Furthermore they implement closures by remembering the outer function's local variables even after the outer function has exited.

```
var v = "top"
var bar
function foo() {
  var v = "foo"
  bar = function() {alert(v)}
}
foo()
bar() // writes "foo", not "top" even though foo() has exited.
```

Objects

For convenience, Types are normally subdivided into *primitives* and *objects*. Objects are entities that have an identity (they are only equal to themselves) and that map property names to values, ("slots" in prototype-based programming terminology). Objects may be thought of as associative arrays or hashes, and are often implemented using these data structures. However, objects have additional features, such as a prototype chain, which ordinary associative arrays do not have.

JavaScript has several kinds of built-in objects, namely Array, Boolean, Date, Function, Math, Number, Object, RegExp and String. Other objects are "host objects", defined not by the language but by the runtime environment. For example, in a browser, typical host objects belong to the DOM (window, form, links etc.).

Creating objects

Objects can be created using a constructor or an object literal. The constructor can use either a built-in Object function or a custom function. It is a convention that constructor functions are given a name that starts with a capital letter:

```
// Constructor
var anObject = new Object();

// Object literal
var objectA = {};
var objectA2 = {};  // A != A2, {}s create new objects as copies.
var objectB = {index1:'value 1', index2:'value 2'};

// Custom constructor (see below)
```

Object literals and array literals allow one to easily create flexible data structures:

```
var myStructure = {
  name: {
    first: "Mel",
    last: "Smith"
  },
  age: 33,
  hobbies: [ "chess", "jogging" ]
};
```

This is the basis for JSON, which is a simple notation that uses JavaScript-like syntax for data exchange.

Methods

A method is simply a function that is assigned to the value of an object's slot. Unlike many object-oriented languages, there is no distinction between a function definition and a method definition. Rather, the distinction occurs during function calling; a function can be called as a method.

When called as a method, the standard local variable *this* is just automatically set to the object instance to the left of the ".". (There are also *call* and *apply* methods that can set *this* explicitly -- some packages such as jQuery do unusual things with *this*.)

In the example below, Foo is being used as a constructor. There is nothing special about a constructor, it is just a method that is invoked after the object is created. *this* is set to the newly created object.

Note that in the example below, Foo is simply assigning values to slots, some of which are functions. Thus it can assign different functions to different instances. There is no prototyping in this example.

```
function y2() {return this.xxx + "2 ";}

function Foo(xz) {
  this.xxx = "yyy-";
  if (xz > 0)
    this.xx = function() {return this.xxx +"X ";};
  else
    this.xx = function() {return this.xxx +"Z ";};
  this.yy = y2;
}

var foo1 = new Foo(1);
var foo2 = new Foo(0);

foo1.y3 = y2; // Assigns the function itself, not its evaluated result,
 i.e. not y2()
foo2.xxx = "aaa-";

alert("foo1/2 " + foo1.xx() + foo2.xx());
// foo1/2 yyy-X aaa-Z

var baz={"xxx": "zzz-"}
baz.y4 = y2 // No need for a constructor to make an object.

alert("yy/y3/y4 " + foo1.yy() + foo1.y3() + baz.y4());
// yy/y3/y4 yyy-2 yyy-2 zzz-2

foo1.y2(); // Throws an exception, because foo1.y2 doesn't exist.
```

Constructors

Constructor functions simply assign values to slots of a newly created object. The values may be data or other functions.

Example: Manipulating an object

```
function MyObject(attributeA, attributeB) {
  this.attributeA = attributeA;
  this.attributeB = attributeB;
}

MyObject.staticC = "blue"; // On MyObject Function, not obj
alert(MyObject.staticC); // blue

obj = new MyObject('red', 1000);

alert(obj.attributeA); // red
```

```
alert(obj["attributeB"]); // 1000

alert(obj.staticC); // undefined

obj.attributeC = new Date(); // add a new property

delete obj.attributeB; // remove a property of obj
alert(obj.attributeB); // undefined

delete obj; // remove the whole Object (rarely used)
alert(obj.attributeA); // throws an exception
```

The constructor itself is stored in the special slot *constructor*. So

```
x = new Foo()
// Above is almost equivalent to
y = {};
y.constructor = Foo;
y.constructor();
// except
x.constructor == y.constructor // true
x instanceof Foo // true
y instanceof Foo // false, surprisingly.
```

Functions are objects themselves, which can be used to produce an effect similar to "static properties" (using C++/Java terminology) as shown below. (The function object also has a special prototype property, as discussed in the Inheritance section below.)

Object deletion is rarely used as the scripting engine will garbage collect objects that are no longer being referenced.

Inheritance

JavaScript supports inheritance hierarchies through prototyping in the manner of Self.

In the following example, the Derive class inherits from the Base class. When d is created as a Derive, the reference to the base instance of Base is copied to d. base.

Derive does not contain a value for aBaseFunction, so it is retrieved from Base *when aBaseFunction is accessed*. This is made clear by changing the value of base.aBaseFunction, which is reflected in the value of d.aBaseFunction.

Some implementations allow the prototype to be accessed or set explicitly using the __proto__ slot as shown below.

```
function Base() {
  this.anOverride = function() {alert("Base::anOverride()");};

  this.aBaseFunction = function() {alert("Base::aBaseFunction()");};
}

function Derive() {
  this.anOverride = function() {alert("Derive::anOverride()");};
}

base = new Base();
```

```
Derive.prototype = base; // Must be before new Derive()

d = new Derive(); // Copies Derive.prototype to d instance's hidden
prototype slot.

base.aBaseFunction = function() {alert("Base::aNEWBaseFunction()")}

d.anOverride(); // Derive::anOverride()
d.aBaseFunction(); // Base::aNEWBaseFunction()
alert(d.aBaseFunction == Derive.prototype.aBaseFunction); // true

alert(d.__proto__ == base); // true in Mozilla-based implementations
but false in many other implementations
```

The following shows clearly how references to prototypes are *copied* on instance creation, but that changes to a prototype can affect all instances that refer to it.

```
function m1() {return "One ";}
function m2() {return "Two ";}
function m3() {return "Three ";}

function Base() {}

Base.prototype.yyy = m2;
bar = new Base();
alert("bar.yyy " + bar.yyy()); // bar.yyy Two

function Top(){this.yyy = m3}
ttt = new Top();

foo = new Base();
Base.prototype = ttt;
  // No effect on foo, the *reference* to ttt is copied.
alert("foo.yyy " + foo.yyy()); // foo.yyy Two

baz = new Base();
alert("baz.yyy " + baz.yyy()); // baz.yyy Three

ttt.yyy = m1; // Does affect baz, and any other derived classes.
alert("baz.yyy1 " + baz.yyy()); // baz.yyy1 One
```

In practice many variations of these themes are used, and it can be both powerful and confusing.

Exceptions

Newer versions of JavaScript (as used in Internet Explorer 5 and Netscape 6) include a try ... catch ... finally exception handling statement to handle run-time errors.

The try ... catch ... finally statement catches exceptions resulting from an error or a throw statement. Its syntax is as follows:

```
try {
  // Statements in which exceptions might be thrown
} catch(errorValue) {
  // Statements that execute in the event of an exception
} finally {
  // Statements that execute afterward either way
}
```

Initially, the statements within the try block execute. If an exception is thrown, the script's control flow immediately transfers to the statements in the catch block, with the exception available as the error argument. Otherwise the catch block is skipped. The Catch block can throw(errorValue) if it does not want to handle a specific error.

In any case the statements in the finally block are always executed. This can be used to free resources, although memory is automatically garbage collected.

Either the catch or the finally clause may be omitted. The catch argument is required.

The Mozilla implementation allows for multiple catch statements, as an extension to the ECMAScript standard. They follow a syntax similar to that used in Java:

```
try { statement; }
catch ( e if e == "InvalidNameException"  ) { statement; }
catch ( e if e == "InvalidIdException"    ) { statement; }
catch ( e if e == "InvalidEmailException" ) { statement; }
catch ( e ) { statement; }
```

In a browser, the onerror event is more commonly used to trap exceptions.

```
function handleErr(errorValue,url,lineNr){...; return true;}
onerror=handleErr;
```

See also

- JavaScript
- ECMAScript
- JScript
- Comparison of Javascript-based source code editors

External links

Reference Material

- David Flanagan, Paula Ferguson: *JavaScript: The Definitive Guide*, O'Reilly & Associates, ISBN 0-596-10199-6
- Danny Goodman: *JavaScript Bible*, Wiley, John & Sons, ISBN 0-7645-3342-8
- Thomas A. Powell, Fritz Schneider: *JavaScript: The Complete Reference*, McGraw-Hill Companies, ISBN 0-07-219127-9
- Emily Vander Veer: *JavaScript For Dummies, 4th Edition*, Wiley, ISBN 0-7645-7659-3

- Mozilla JavaScript Language Documentation [6]
- A re-introduction to JavaScript - Mozilla Developer Center [7]
- Interactive JavaScript Lessons - example-based [8]
- JavaScript on About.com: lessons and explanation [9]
- ECMAScript standard references: ECMA-262 [10]
- Mozilla Developer Center Core References for JavaScript versions 1.5 [11], 1.4 [12], 1.3 [13] and 1.2 [14]

References

[1] http://hepunx.rl.ac.uk/~adye/jsspec11/intro.htm#1006028

[2] Flanagan, David (2006). *JavaScript: The definitive Guide*. p. 16. ISBN 978-0-596-10199-2. "Omitting semicolons is not a good programming practice; you should get into the habit of inserting them."

[3] http://www.jibbering.com/faq/#numbers

[4] http://www.jibbering.com/faq/#formatNumber

[5] The issue is whether it would iterate not only through the array indices, but also through other visible properties.

An article in the Microsoft Developer Network website (http://msdn.microsoft.com/en-us/library/kw1tezhk(VS.85).aspx) specifically states that For...In can be used for *"stepping through ... all the elements of an array"*. The MSDN article refers to JScript, which is, effectively, what is used by Internet Explorer and Windows Script Host for JavaScript script. An example in the W3Schools website (http://www.w3schools.com/js/js_loop_for_in.asp) gives arrays as an example of the use of For...In.

An article in the Mozilla Developer Centre (https://developer.mozilla.org/en/Core_JavaScript_1.5_Guide/Object_Manipulation_Statements#for...in_Statement) explains more about the problem: *"Although it may be tempting to use this as a way to iterate over Array elements, because the for...in statement iterates over user-defined properties in addition to the array elements, if you modify the Array object, such as adding custom properties or methods, the for...in statement will return the name of your user-defined properties in addition to the numeric indexes. Thus it is better to use a traditional for loop with a numeric index when iterating over arrays."*

[6] https://developer.mozilla.org/en/docs/JavaScript

[7] https://developer.mozilla.org/en/docs/A_re-introduction_to_JavaScript

[8] http://javalessons.com/cgi-bin/fun/java-tutorials-main.cgi?sub=javascript&code=script

[9] http://javascript.about.com/

[10] http://www.ecma-international.org/publications/standards/Ecma-262.htm

[11] https://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference

[12] http://research.nihonsoft.org/javascript/CoreReferenceJS14/

[13] http://research.nihonsoft.org/javascript/ClientReferenceJS13/

[14] http://research.nihonsoft.org/javascript/jsref/

Article Sources and Contributors

JavaScript syntax *Source*: http://en.wikipedia.org/w/index.php?oldid=360082398 *Contributors*: (jarbarf), Aaaidan, Acasson, Albrecht andrzejewski, AnAj, Ancheta Wis, Arthur Rubin, Betax, Buzzard2501, Cactus.man, Chutzpan, Clayhalliwell, Cliciu, Cmcfarland, Comocomocomocomo, Crystallina, David-Sarah Hopwood, Davidfstr, Dipset1991, Dougluce, Dreftymac, Drknkn, Dto, Enkidu1947, Erich gasboy, Exert, Explanator, Extremecircuitz, Flash200, FrankSier, Fred Bradstadt, Furrykef, Gabrielsroka, GavinSharp, Gerbrant, Grahamzibar, Grshiplett, Imshardy, Inimino, Insouciance, Int19h, J.delanoy, JLaTondre, Jeffz1, Jeremywosborne, Jeshan, Jessemerriman, Jiri Svoboda, John Vandenberg, Jorge Stolfi, KMeyer, Kgwikipedian, Khaledziyaeen, Kickboy, Kohtala, Kusunose, Lanxiazhi, Lingwitt, Lucian1900, Luna Santin, LuoShengli, Lycurgus, Machine Elf 1735, Maian, Mattjball, Mikeblas, Mild Bill Hiccup, Misortie, Mmalessa, Nate879, Nigelj, OverlordQ, Piano non troppo, PleaseStand, Quamaretto, Quilokos, RHaworth, Reazal, RobG-bne, Rockower, Rsjaffe, Rufous, Rursus, Ruud Koot, Sesembeki, Shanes, Sleeper220, Takarada, ThomasStrohmann, Thumperward, Tobias Bergemann, Tony Sidaway, Triul, Tryforceful, Tuntable, Viebel, Voyagerfan5761, Walden, Wknight94, Wogga62, X42bn6, Ynhockey, Youngoat, Zzedar, 257 anonymous edits

Image Sources, Licenses and Contributors

Image:Symbol note.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Symbol_note.svg *License*: Public Domain *Contributors*: User:KyraVixen

License