WIKIPEDIA

JavaScript syntax

The **syntax of JavaScript** is the set of rules that define a correctly structured JavaScript program.

The examples below make use of the log function of the console object present in most browsers for standard text output.

The JavaScript standard library lacks an official standard text output function. Given that JavaScript is mainly used for client-side scripting within modern Web browsers, and that almost all Web browsers provide the alert function, `alert` can also be used, but is not commonly used.

Contents

Origins

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

Basics

Case sensitivity

JavaScript is case sensitive. It is common to start the name of a constructor with a capitalised letter, and the name of a function or variable with a lower-case letter.

Example:

```
let a=5;
console.log(a); // 5
console.log(A); // throws a ReferenceError: A is not defined
```

Whitespace and semicolons

Spaces, tabs and newlines used outside of string constants are called whitespace. Unlike in C, whitespace in JavaScript source can directly impact semantics. Because of a technique called "automatic semicolon insertion" (ASI), 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). Some authorities advise supplying statement-terminating semicolons explicitly, because it may lessen unintended effects of the automatic semicolon insertion.[2]

There are two issues: five tokens can either begin a statement or be the extension of a complete statement; and five restricted productions, where line breaks are not allowed in certain positions, potentially yielding incorrect parsing.[3]

The five problematic tokens are the open parenthesis "`(`", open bracket "`[`", slash "`/`", plus "`+`", and minus "`-`". Of these, the open parenthesis is common in the immediately-invoked function expression pattern, and open bracket occurs sometimes, while others are quite rare. The example given in the spec is:[3]

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

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

with the suggestion that the preceding statement be terminated with a semicolon.

Some suggest instead the use of *leading* semicolons on lines starting with '`(`' or '`[`', so the line is not accidentally joined with the previous one. This is known as a **defensive semicolon**, and is particularly recommended, because code may otherwise become ambiguous when it is rearranged.[3][4] For example:

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

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

Initial semicolons are also sometimes used at the start of JavaScript libraries, in case they are appended to another library that omits a trailing semicolon, as this can result in ambiguity of the initial statement.

The five restricted productions are `return`, `throw`, `break`, `continue`, and post-increment/decrement. In all cases, inserting semicolons does not fix the problem, but makes the parsed syntax clear, making the error easier to detect. `return` and `throw` take an optional value, while `break` and `continue` take an optional label. In all cases, the advice is to keep the value or label on the same line as the statement. This most often shows up in the return statement, where one might return a large object literal, which might be accidentally placed starting on a new line. For post-increment/decrement, there is potential ambiguity with pre-increment/decrement, and again it is recommended to simply keep these on the same line.

```
return
a + b;

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

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. */

/* Comments /* may not be nested */ Syntax error */
```

Variables

Variables in standard JavaScript have no type attached, and any value can be stored in any variable. Starting with ES6, the version of the language finalised in 2015, variables can be declared with `let` (for a block level variable), `var` (for a function level variable) or `const` (for an immutable one). However, while the object assigned to a `const` cannot be changed, its properties can. Before ES6, variables were declared only with a `var` statement. An identifier must start with a letter, underscore (_), or dollar sign ($); 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).

Starting with JavaScript 1.5, ISO 8859-1 or Unicode letters (or `\uXXXX` Unicode escape sequences) can be used in identifiers.[5] In certain JavaScript implementations, the at sign (@) can be used in an identifier, but this is contrary to the specifications and not supported in newer implementations.

Scoping and hoisting

Variables are lexically scoped at function level (not block level as in C), and this does not depend on order (forward declaration is not necessary): if a variable is declared inside a function (at any point, in any block), then inside the function, the name will resolve to that variable. This is equivalent in block scoping to variables being forward declared at the top of the function, and is referred to as *hoisting*.[6]

However, the variable value is `undefined` until it is initialized, and forward reference is not possible. Thus a `var x = 1` statement in the middle of the function is equivalent to a `var x` declaration statement at the top of the function, and an `x = 1` assignment statement at that point in the middle of the function – only the declaration is hoisted, not the assignment.

Function statements, whose effect is to declare a variable of type `Function` and assign a value to it, are similar to variable statements, but in addition to hoisting the declaration, they also hoist the assignment – as if the entire statement appeared at the top of the containing function – and thus forward reference is also possible: the location of a function statement within an enclosing function is irrelevant.

Be sure to understand that

```
var func = function() { .. } // will NOT be hoisted
function func() { .. } // will be hoisted
```

Block scoping can be produced by wrapping the entire block in a function and then executing it — this is known as the immediately-invoked function expression pattern — or by declaring the variable using the `let` keyword.

Declaration and assignment

Variables declared outside any function are global. If a variable is declared in a higher scope, it can be accessed by child functions.

When JavaScript tries to **resolve** an identifier, it looks in the local function scope. If this identifier is not found, it looks in the outer function that declared the local one, and so on along the *scope chain* until it reaches the *global scope* where global variables reside. If it is still not found, JavaScript will raise a `ReferenceError` exception.

When **assigning** an identifier, JavaScript goes through exactly the same process to retrieve this identifier, except that if it is not found in the *global scope*, it will create the "variable" as a property of the *global object*.[7] As a consequence, a variable never declared will be global, if assigned. Declaring a variable (with the keyword `var`) in the *global code* (i.e. outside of any function body), assigning a never declared identifier or adding a property to the *global object* (usually *window*) will also create a new global variable.

Note that JavaScript's *strict mode* forbids the assignment of an undeclared variable, which avoids global namespace pollution. Also `const` cannot be declared without initialization.

Examples

Here are examples of variable declarations and scope:

```
var x = 0; // A global variable, because it is not in any function

function f() {
  var z = 'foxes', r = 'birds'; // 2 local variables
  m = 'fish'; // global, because it wasn't declared anywhere before

  function child() {
    var r = 'monkeys'; // This variable is local and does not affect the "birds" r of the parent function.
    z = 'penguins'; // Closure: Child function is able to access the variables of the parent function.
  }

  twenty = 20; // This variable is declared on the next line, but usable anywhere in the function, even before, as
here
  var twenty;

  child();
  return x; // We can use x here, because it is global
}

f();

console.log(z); // This line will raise a ReferenceError exception, because the value of z is no longer available
```

```
for (let i=0;i<10;i++) console.log(i);
console.log(i); // throws a ReferenceError: i is not defined
```

```
for (const i=0;i<10;i++) console.log(i); // throws a TypeError: Assignment to constant variable

const pi; // throws a SyntaxError: Missing initializer in const declaration
```

Primitive data types

The JavaScript language provides six primitive data types:

- Undefined
- Null
- Number
- String
- Boolean
- Symbol

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.

Note: undefined is considered a genuine primitive type. Unless explicitly converted, the undefined value may behave unexpectedly in comparison to other types that evaluate to false in a logical context.

```
var test;                         // variable declared, but not defined, ...
                                  // ... set to value of undefined
var testObj = {};
console.log(test);                // test variable exists, but value not ...
                                  // ... defined, displays undefined
console.log(testObj.myProp);      // testObj exists, property does not, ...
                                  // ... displays undefined
console.log(undefined == null);   // unenforced type during check, displays true
console.log(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 in versions before ECMAScript 5, it is legal for someone to write `var undefined = "I'm defined now";`. A more robust approach is to compare using `(typeof x === 'undefined')`.

Functions like this won't work as expected:

```
function isUndefined(x) { var u; return x === u; }             // like this...
function isUndefined(x) { return x === void 0; }               // ... or that second one
function isUndefined(x) { return (typeof x) === "undefined"; } // ... or that third one
```

Here, calling `isUndefined(my_var)` raises a `ReferenceError` if `my_var` is an unknown identifier, whereas `typeof my_var === 'undefined'` doesn't.

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. Surprisingly, `null` is considered an object by `typeof`.

```
console.log(null == undefined);        // unenforced type during check, displays true
console.log(null === undefined);       // enforce type during check, displays false
console.log(typeof null === 'object'); // true
```

Number

Numbers are represented in binary as IEEE-754 doubles, which provides an accuracy of nearly 16 significant digits. Because they are floating point numbers, they do not always exactly represent real numbers, including fractions.

This becomes an issue when comparing or formatting numbers. For example:

```
console.log(0.2 + 0.1 == 0.3); // displays true as per ECMASCRIPT  6 Specifications
console.log(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 (http://www.jibbering.com/faq/#formatNumber).

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
0b1011; // a binary integer equal to 11
0o377;   // 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
```

The extents **+∞**, **−∞** and **NaN** (Not a Number) of the number type may be obtained by two program expressions:

```
Infinity; // positive infinity (negative obtained with -Infinity for instance)
NaN;      // The Not-A-Number value, also returned as a failure in ...
          // ... string-to-number conversions
```

Infinity and NaN are numbers:

```
typeof Infinity;   // returns "number"
typeof NaN;        // returns "number"
```

These three special values correspond and behave as the IEEE-754 describes them.

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);
```

However, it's impossible to use equality operators (== and ===) to determine whether a value is NaN:

```
console.log(NaN == NaN);   // false
console.log(NaN === NaN);  // false
console.log(isNaN(NaN));   // true
```

String

A string in JavaScript is a sequence of characters. In JavaScript, strings can be created directly (as literals) by placing the series of characters between double (") or single (') quotes. Such strings must be written on a single line, but may include escaped newline characters (such as \n). The JavaScript standard allows the backquote character (`, a.k.a. grave accent or backtick) to quote multiline literal strings, but this is supported only on certain browsers as of 2016: Firefox and Chrome, but not Internet Explorer 11.[8]

```
var greeting = "Hello, World!";
var anotherGreeting = 'Greetings, people of Earth.';
```

Individual characters within a string can be accessed using the `charAt` method (provided by `String.prototype`). This is the preferred way when accessing individual characters within a string, because it also works in non-modern browsers:

```
var h = greeting.charAt(0);
```

In modern 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];
```

However, JavaScript strings are immutable:

```
greeting[0] = "H"; // Fails.
```

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

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

Quotes of the same type cannot be nested 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."; // Works by escaping " with \"
```

It is possible to create a string using the `String` constructor:

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

These objects have a `valueOf` method returning the primitive string wrapped within them:

```
var s = new String("Hello !");
typeof s; // Is 'object'.
typeof s.valueOf(); // Is 'string'.
```

Equality between two `String` objects does not behave as with string primitives:

```
var s1 = new String("Hello !");
var s2 = new String("Hello !");
s1 == s2; // Is false, because they are two distinct objects.
s1.valueOf() == s2.valueOf(); // Is true.
```

Boolean

JavaScript provides a Boolean data type with `true` and `false` literals. The `typeof` operator returns the string `"boolean"` for these primitive types. When used in a logical context, `0`, `-0`, `null`, `NaN`, `undefined`, and the empty string (`""`) evaluate as `false` due to automatic type coercion. All other values (the complement of the previous list) evaluate as `true`, including the strings `"0"`, `"false"` and any object. Automatic type coercion by the equality comparison operators (`==` and `!=`) can be avoided by using the type checked comparison operators (`===` and `!==`).

When type conversion is required, JavaScript converts `Boolean`, `Number`, `String`, or `Object` operands as follows:[9]

Number and String
: The string is converted to a number value. JavaScript attempts to convert the string numeric literal to a Number type value. First, a mathematical value is derived from the string numeric literal. Next, this value is rounded to nearest Number type value.

Boolean
: If one of the operands is a Boolean, the Boolean operand is converted to 1 if it is `true`, or to 0 if it is `false`.

Object
: If an object is compared with a number or string, JavaScript attempts to return the default value for the object. An object is converted to a primitive String or Number value, using the `.valueOf()` or `.toString()` methods of the object. If this fails, a runtime error is generated.

Douglas Crockford advocates the terms "truthy" and "falsy" to describe how values of various types behave when evaluated in a logical context, especially in regard to edge cases.[10] The binary logical operators returned a Boolean value in early versions of JavaScript, but now they return one of the operands instead. The left–operand is returned, if it can be evaluated as : `false`, in the case of conjunction: (`a && b`), or `true`, in the case of disjunction: (`a || b`); otherwise the right–operand is returned. Automatic type coercion by the comparison operators may differ for cases of mixed Boolean and number-compatible operands (including strings that can be evaluated as a number, or objects that can be evaluated as such a string), because the Boolean operand will be compared as a numeric value. This may be unexpected. An expression can be explicitly cast to a Boolean primitive by doubling the logical negation operator: (`!!`), using the `Boolean()` function, or using the conditional operator: (`c ? t : f`).

```
// Automatic type coercion
console.log(true  ==   2 ); // false... true  → 1 !== 2 ←  2
console.log(false ==   2 ); // false... false → 0 !== 2 ←  2
console.log(true  ==   1 ); // true.... true  → 1 === 1 ←  1
console.log(false ==   0 ); // true.... false → 0 === 0 ←  0
console.log(true  ==  "2"); // false... true  → 1 !== 2 ← "2"
console.log(false ==  "2"); // false... false → 0 !== 2 ← "2"
console.log(true  ==  "1"); // true.... true  → 1 === 1 ← "1"
console.log(false ==  "0"); // true.... false → 0 === 0 ← "0"
console.log(false ==  "" ); // true.... false → 0 === 0 ← ""
console.log(false ==  NaN); // false... false → 0 !== NaN

console.log(NaN == NaN); // false...... NaN is not equivalent to anything, including NaN.

// Type checked comparison (no conversion of types and values)
console.log(true === 1); // false...... data types do not match

// Explicit type coercion
console.log(true === !!2);   // true.... data types and values match
console.log(true === !!0);   // false... data types match, but values differ
console.log( 1  ? true : false); // true.... only ±0 and NaN are “falsy” numbers
console.log("0" ? true : false); // true.... only the empty string is “falsy”
console.log(Boolean({}));    // true.... all objects are “truthy”
```

The new operator can be used to create an object wrapper for a Boolean primitive. However, the `typeof` operator does not return `boolean` for the object wrapper, it returns `object`. Because all objects evaluate as `true`, a method such as `.valueOf()`, or `.toString()`, must be used to retrieve the wrapped value. For explicit coercion to the Boolean type, Mozilla recommends that the `Boolean()` function (without `new`) be used in preference to the Boolean object.

```
var b = new Boolean(false);   // Object  false {}
var t = Boolean(b);           // Boolean true
var f = Boolean(b.valueOf()); // Boolean false
var n = new Boolean(b);       // Not recommended
n = new Boolean(b.valueOf()); // Preferred

if (0 || -0 || "" || null || undefined || b.valueOf() || !new Boolean() || !t) {
  console.log("Never this");
} else if ([] && {} && b && typeof b === "object" && b.toString() === "false") {
  console.log("Always this");
}
```

Symbol

New in ECMAScript6. A *Symbol* is a unique and immutable identifier.

Example:

```
x=Symbol(1);
y=Symbol(1);
x==y; // false
arr=[x,y];
arr[x]=1;
arr[y]=2; // x and y are unique keys for the array arr
arr[x]; // displays 1
arr[y]; // displays 2
x=Symbol(3);
arr; // displays [Symbol(1),Symbol(1)]
arr[x]; // is now undefined
x=Symbol(1);
arr[x]; // undefined
```

The Symbol wrapper also provides access to a variable free iterator.

```
x=[1,2,3,4]; // x is an Array and iterable
ex=x[Symbol.iterator](); // provides an iterator for x
while ((exv=ex.next().value)!=undefined) console.log(exv); // displays 1,2,3,4
```

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 JavaScript object prototyped from the `Array` constructor 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 (for example, `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
console.log(myArray[0]);           // Equivalent to console.log("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 `Array`s may be accessed using normal object property access notation:

```
myArray[1];  // 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 defined elements 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.

One 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"]; // results in "brown"
dog.color;    // also results in "brown"
```

One can use the object and array declaration literals to quickly create arrays that are associative, multidimensional, or both. (Technically, JavaScript does not support multidimensional arrays, but one can mimic them with arrays-of-arrays.)

```
cats = [{color: "brown", size: "large"},
    {color: "black", size: "small"}];
cats[0]["size"];      // results in "large"

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

Date

A `Date` object stores a signed millisecond count with zero representing 1970-01-01 00:00:00 UT and a range of ±10^8 days. There are several ways of providing arguments to the `Date` constructor. Note that months are zero-based.

```
new Date();                       // create a new Date instance representing the current time/date.
new Date(2010, 2, 1);             // create a new Date instance representing 2010-Mar-01 00:00:00
new Date(2010, 2, 1, 14, 25, 30); // create a new Date instance representing 2010-Mar-01 14:25:30
new Date("2010-3-1 14:25:30");    // create a new Date instance from a String.
```

Methods to extract fields are provided, as well as a useful `toString`:

```
var d = new Date(2010, 2, 1, 14, 25, 30); // 2010-Mar-01 14:25:30;

// Displays '2010-3-1 14:25:30':
console.log(d.getFullYear() + '-' + (d.getMonth() + 1) + '-' + d.getDate() + ' '
    + d.getHours() + ':' + d.getMinutes() + ':' + d.getSeconds());

// Built-in toString returns something like 'Mon Mar 01 2010 14:25:30 GMT-0500 (EST)':
console.log(d);
```

Error

Custom error messages can be created using the `Error` class:

```
throw new Error("Something went wrong.");
```

These can be caught by try...catch...finally blocks as described in the section on exception handling.

Math

The `Math` object contains various math-related constants (for example, π) and functions (for example, cosine). (Note that the `Math` object has no constructor, unlike `Array` or `Date`. All its methods are "static", that is "class" methods.) All the trigonometric functions use angles expressed in radians, not degrees or grads.

Properties of the Math object

Property	Returned value rounded to 5 digits	Description
`Math.E`	2.7183	e: Natural logarithm base
`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	Returned value 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 ($-\pi/2$ to $+\pi/2$)
`Math.atan2(-3.7, -3.7)`	-2.3562 rad. = -135°	Whole circle arctangent ($-\pi$ to $+\pi$)
`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);     // returns Boolean
"string".search(/expression/); // returns position Number
"string".replace(/expression/, replacement);

// Here are some examples
if (/Tom/.test("My name is Tom")) console.log("Hello Tom!");
console.log("My name is Tom".search(/Tom/));          // == 11 (letters before Tom)
console.log("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'))                   console.log('Digit');
if (/[0-9]/.test('6'))                console.log('Digit');
if (/[13579]/.test('1'))              console.log('Odd number');
if (/\S\S\s\S\S\S\S/.test('My name')) console.log('Format OK');
if (/\w\w\w/.test('Tom'))             console.log('Hello Tom');
if (/[a-zA-Z]/.test('B'))             console.log('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')) console.log ('Hi Tom, Tam or Tim');
if (/A|B/.test("A"))  console.log ('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"))       console.log("OK"); // match: "ac", "abc"
if (/ab*c/.test("ac"))       console.log("OK"); // match: "ac", "abc", "abbc", "abbbc" etc.
if (/ab+c/.test("abc"))      console.log("OK"); // match: "abc", "abbc", "abbbc" etc.
if (/ab{3}c/.test("abbbc"))  console.log("OK"); // match: "abbbc"
if (/ab{3,}c/.test("abbbc")) console.log("OK"); // match: "abbbc", "abbbbc", "abbbbbc" etc.
if (/ab{1,3}c/.test("abc"))  console.log("OK"); // match: "abc", "abbc", "abbbc"
```

Anchors

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

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

Subexpression

```
// ( )  - groups characters

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

Flags

```
// /g  - global
// /i  - ignore upper/lower case
// /m  - allow matches to span multiple lines

console.log("hi tom!".replace(/Tom/i, "John"));  // == "hi John!"
console.log("ratatam".replace(/ta/, "tu"));      // == "ratutam"
console.log("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/g); // 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) {
  console.log("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"
  console.log("It is " + my_time + " on " + my_date);
} else console.log("Did not find a valid date!");
```

Function

Every function in JavaScript is an instance of the `Function` constructor:

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

The add function above may also be defined using a function expression:

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

There exists a shorthand for assigning a function expression to a variable, and is as follows:

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

Or

```
var add = ((x, y) => {
  return x + y;
});
// or
var add = ((x, y) => x + y);

var t = add(1, 2);
console.log(t); // 3
```

A function instance has properties and methods.

```
function subtract(x, y) {
  return x - y;
}

console.log(subtract.length); // 2, expected amount of arguments.
console.log(subtract.toString());

/*
"function subtract(x, y) {
  return x - y;
}"
*/
```

Operators

The '+' operator is overloaded: it is used for string concatenation and arithmetic addition. This may cause problems when inadvertently mixing strings and numbers. As a unary operator, it can convert a numeric string to a number.

```
// Concatenate 2 strings
console.log('He' + 'llo'); // displays Hello

// Add two numbers
console.log(2 + 6);  // displays 8

// Adding a number and a string results in concatenation
console.log(2 + '2');    // displays 22
console.log('$' + 3 + 4);  // displays $34, but $7 may have been expected
console.log('$' + (3 + 4)); // displays $7
console.log(3 + 4 + '7'); // displays 77, numbers stay numbers until a string is added

// Convert a string to a number
console.log(+'2' === 2); // displays true
console.log(+'Hello'); // displays NaN
```

Similarly, the '*' operator is overloaded: it can convert a string into a number.

```
console.log(2 + '6'*1);  // displays 8
console.log(3*'7'); // 21
console.log('3'*'7'); // 21
console.log('hello'*'world'); // displays NaN
```

Arithmetic

JavaScript supports the following **binary arithmetic operators**:

+	addition
-	subtraction
*	multiplication
/	division (returns a floating-point value)
%	modulo (returns the remainder)
**	exponentiation

JavaScript supports the following **unary arithmetic 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;
console.log(++x); // x becomes 2; displays 2
console.log(x++); // displays 2; x becomes 3
console.log(x);  // x is 3; displays 3
console.log(x--); //  displays 3; x becomes 2
console.log(x);  //  displays 2; x is 2
console.log(--x); //  x becomes 1; displays 1
```

The modulo operator displays the remainder after division by the modulus. If negative numbers are involved, the returned value depends on the operand.

```
var x = 17;
console.log(x%5); // displays 2
console.log(x%6); // displays 5
console.log(-x%5); // displays -2
console.log(-x%-5); // displays -2
console.log(x%-5); // displays 2
```

To always return a non-negative number, re-add the modulus and apply the modulo operator again:

```
var x = 17;
console.log((-x%5+5)%5); // displays 3
```

Assignment

=	assign
+=	add and assign
-=	subtract and assign
*=	multiply and assign
/=	divide and assign
%=	modulo and assign
**=	exponentiation and assign

Assignment of primitive types

```
var x = 9;
x += 1;
console.log(x); // displays: 10
x *= 30;
console.log(x); // displays: 300
x /= 6;
console.log(x); // displays: 50
x -= 3;
console.log(x); // displays: 47
x %= 7;
console.log(x); // displays: 5
```

Assignment of object types

```
/**
 * To learn JavaScript objects...
 */
var object_1 = {a: 1};      // assign reference of newly created object to object_1
var object_2 = {a: 0};
var object_3 = object_2;    // object_3 references the same object as object_2 does

object_3.a = 2;
message();              // displays 1 2 2

object_2 = object_1;        // object_2 now references the same object as object_1
                        // object_3 still references what object_2 referenced before
message();              // displays 1 1 2

object_2.a = 7;             // modifies object_1
message();              // displays 7 7 2

object_3.a = 5;                 // object_3 doesn't change object_2
message();                  // displays 7 7 5

object_3 = object_2;
object_3.a=4;                  // object_3 changes object_1 and object_2
message();                     // displays 4 4 4

/**
 * Prints the console.log message
 */
function message() {
    console.log(object_1.a + " " + object_2.a + " " + object_3.a);
}
```

Destructuring assignment

In Mozilla's JavaScript, since version 1.7, destructuring assignment allows the assignment of parts of data structures to several variables at once. The left hand side of an assignment is a pattern that resembles an arbitrarily nested object/array literal containing l-lvalues at its leaves that are to receive the substructures of the assigned value.

```
var a, b, c, d, e;
[a, b, c] = [3, 4, 5];
console.log(a + ',' + b + ',' + c); // displays: 3,4,5
e = {foo: 5, bar: 6, baz: ['Baz', 'Content']};
var arr = [];
({baz: [arr[0], arr[3]], foo: a, bar: b}) = e;
console.log(a + ',' + b + ',' + arr);   // displays: 5,6,Baz,,,Content
[a, b] = [b, a];        // swap contents of a and b
console.log(a + ',' + b);       // displays: 6,5

[a, b, c] = [3, 4, 5]; // permutations
[a, b, c] = [b, c, a];
console.log(a + ',' + b + ',' + c); // displays: 4,5,3
```

Spread/rest operator

The ECMAScript 2015 standard introduces the "`...`" operator, for the related concepts of "spread syntax"[11] and "rest parameters"[12]

Spread syntax provides another way to destructure arrays. It indicates that the elements in a specified array should be used as the parameters in a function call or the items in an array literal.

In other words, "`...`" transforms "`[...foo]`" into "`[foo[0], foo[1], foo[2]]`", and "`this.bar(...foo);`" into "`this.bar(foo[0], foo[1], foo[2]);`".

```
var a = [1, 2, 3, 4];

// It can be used multiple times in the same expression
var b = [...a, ...a]; // b = [1, 2, 3, 4, 1, 2, 3, 4];

// It can be combined with non-spread items.
var c = [5, 6, ...a, 7, 9]; // c = [5, 6, 1, 2, 3, 4, 7, 9];

// For comparison, doing this without the spread operator
// creates a nested array.
var d = [a, a]; // d = [[1, 2, 3, 4], [1, 2, 3, 4]]

// It works the same with function calls
function foo(arg1, arg2, arg3) {
    console.log(arg1 + ':' + arg2 + ':' + arg3);
}

// You can use it even if it passes more parameters than the function will use
foo(...a); // "1:2:3" → foo(a[0], a[1], a[2], a[3]);

// You can mix it with non-spread parameters
foo(5, ...a, 6); // "5:1:2" → foo(5, a[0], a[1], a[2], a[3], 6);

// For comparison, doing this without the spread operator
// assigns the array to arg1, and nothing to the other parameters.
foo(a); // "1,2,3,4:undefined:undefined"
```

When `...` is used in a function *declaration*, it indicates a **rest parameter**. The rest parameter must be the final named parameter in the function's parameter list. It will be assigned an `Array` containing any arguments passed to the function in excess of the other named parameters. In other words, it gets "the rest" of the arguments passed to the function (hence the name).

```
function foo(a, b, ...c) {
    console.log(c.length);
}

foo(1, 2, 3, 4, 5); // "3" → c = [3, 4, 5]
foo('a', 'b'); // "0" → c = []
```

Rest parameters are similar to Javascript's `arguments` object, which is an array-like object that contains all of the parameters (named and unnamed) in the current function call. Unlike `arguments`, however, rest parameters are true `Array` objects, so methods such as `.slice()` and `.sort()` can be used on them directly.

The `...` operator can only be used with `Array` objects. (However, there is a proposal to extend it to `Objects` in a future ECMAScript standard.[13])

Comparison

==	equal
!=	not equal
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
===	identical (equal and of same type)
!==	not identical

Variables referencing objects are equal or identical only if they reference the same object:

```
var obj1 = {a: 1};
var obj2 = {a: 1};
var obj3 = obj1;
console.log(obj1 == obj2);  //false
console.log(obj3 == obj1);  //true
console.log(obj3 === obj1); //true
```

See also String.

Logical

JavaScript provides four logical operators:

- unary negation (`NOT = !a`)
- binary disjunction (`OR = a || b`) and conjunction (`AND = a && b`)
- ternary conditional (`c ? t : f`)

In the context of a logical operation, any expression evaluates to true except the following**:**

- Strings: `""`, `''`,
- Numbers: `0`, `-0`, `NaN`,

- Special: `null`, `undefined`,
- Boolean: `false`.

The Boolean function can be used to explicitly convert to a primitive of type `Boolean`:

```
// Only empty strings return false
console.log(Boolean("")      === false);
console.log(Boolean("false") === true);
console.log(Boolean("0")     === true);

// Only zero and NaN return false
console.log(Boolean(NaN) === false);
console.log(Boolean(0)   === false);
console.log(Boolean(-0)  === false); // equivalent to -1*0
console.log(Boolean(-2)  === true);

// All objects return true
console.log(Boolean(this) === true);
console.log(Boolean({})   === true);
console.log(Boolean([])   === true);

// These types return false
console.log(Boolean(null)      === false);
console.log(Boolean(undefined) === false); // equivalent to Boolean()
```

The NOT operator evaluates its operand as a Boolean and returns the negation. Using the operator twice in a row, as a double negative, explicitly converts an expression to a primitive of type Boolean:

```
console.log( !0 === Boolean(!0));
console.log(Boolean(!0) === !!1);
console.log(!!1 === Boolean(1));
console.log(!!0 === Boolean(0));
console.log(Boolean(0) === !1);
console.log(!1 === Boolean(!1));
console.log(!"" === Boolean(!""));
console.log(Boolean(!"") === !!"s");
console.log(!!"s" === Boolean("s"));
console.log(!!"" === Boolean(""));
console.log(Boolean("") === !"s");
console.log(!"s" === Boolean(!"s"));
```

The ternary operator can also be used for explicit conversion:

```
console.log([] == false); console.log([] ? true : false); // “truthy”, but the comparison uses [].toString()
console.log([0] == false); console.log([0]? true : false); // [0].toString() == "0"
console.log("0" == false); console.log("0"? true : false); // "0" → 0 ... (0 == 0) ... 0 ← false
console.log([1] == true); console.log([1]? true : false); // [1].toString() == "1"
console.log("1" == true); console.log("1"? true : false); // "1" → 1 ... (1 == 1) ... 1 ← true
console.log([2] != true); console.log([2]? true : false); // [2].toString() == "2"
console.log("2" != true); console.log("2"? true : false); // "2" → 2 ... (2 != 1) ... 1 ← true
```

Expressions that use features such as post–incrementation (`i++`) have an anticipated side effect. JavaScript provides short-circuit evaluation of expressions; the right operand is only executed if the left operand does not suffice to determine the value of the expression.

```
console.log(a || b);  // When a is true, there is no reason to evaluate b.
console.log(a && b);  // When a is false, there is no reason to evaluate b.
console.log(c ? t : f); // When c is true, there is no reason to evaluate f.
```

In early versions of JavaScript and JScript, the binary logical operators returned a Boolean value (like most C-derived programming languages). However, all contemporary implementations return one of their operands instead:

```
console.log(a || b); // if a is true, return a, otherwise return b
console.log(a && b); // if a is false, return a, otherwise return b
```

Programmers who are more familiar with the behavior in C might find this feature surprising, but it allows for a more concise expression of patterns like null coalescing:

```
var s = t || "(default)"; // assigns t, or the default value, if t is null, empty, etc.
```

Bitwise

JavaScript supports the following **binary bitwise operators**:

&	AND
|	OR
^	XOR
!	NOT
<<	shift left (zero fill at right)
>>	shift right (sign-propagating); copies of the leftmost bit (sign bit) are shifted in from the left
>>>	shift right (zero fill at left). For positive numbers, >> and >>> yield the same result.

Examples:

```
x=11 & 6;
console.log(x); // 2
```

JavaScript supports the following **unary bitwise operator**:

~	NOT (inverts the bits)

Bitwise Assignment

JavaScript supports the following **binary assignment operators:**

&=	and
|=	or
^=	xor
<<=	shift left (zero fill at right)
>>=	shift right (sign-propagating); copies of the leftmost bit (sign bit) are shifted in from the left
>>>=	shift right (zero fill at left). For positive numbers, >>= and >>>= yield the same result.

Examples:

```
x=7;
console.log(x); // 7
x<<=3;
console.log(x); // 7->14->28->56
```

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 (ternary) 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.

```
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

The syntax of the JavaScript switch statement is as follows:

```
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.
- String literal values can also be used for the case values.
- Expressions can be used instead of values.
- The default case (optional) is executed when the expression does not match any other specified cases.
- Braces are required.

For loop

The syntax of the JavaScript for loop is as follows:

```
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(iteration)) // one statement
```

For ... in loop

The syntax of the JavaScript `for ... in loop` is as follows:

```
for (var property_name in some_object) {
  // statements using some_object[property_name];
}
```

- Iterates through all enumerable properties of an object.
- Iterates through all used indices of array including all user-defined properties of array object, if any. Thus it may be better to use a traditional for loop with a numeric index when iterating over arrays.
- 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. It is useful to test for a given property using `if (some_object.hasOwnProperty(property_name)) { ...}`. Thus, adding a method to the array prototype with `Array.prototype.newMethod = function() {...}` may cause `for ... in` loops to loop over the method's name.

While loop

The syntax of the JavaScript while loop is as follows:

```
while (condition) {
  statement1;
  statement2;
  statement3;
  ...
}
```

Do ... while loop

The syntax of the JavaScript `do ... while loop` is as follows:

```
do {
  statement1;
  statement2;
  statement3;
  ...
} while (condition);
```

With

The with statement adds all of the given object's properties and methods into the following block's scope, letting them be referenced as if they were local variables.

```
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.

Because the availability of with statements hinders program performance and is believed to reduce code clarity (since any given variable could actually be a property from an enclosing `with`), this statement is not allowed in *strict mode*.

Labels

JavaScript supports nested labels in most implementations. Loops or blocks can be labelled for the break statement, and loops for `continue`. Although `goto` is a reserved word,[14] `goto` is not implemented in JavaScript.

```
loop1: for (var a = 0; a < 10; a++) {
  if (a == 4) {
    break loop1; // Stops after the 4th attempt
  }
  console.log('a = ' + a);
  loop2: for (var b = 0; b < 10; ++b) {
    if (b == 3) {
     continue loop2; // Number 3 is skipped
    }
    if (b == 6) {
     continue loop1; // Continues the first loop, 'finished' is not shown
    }
    console.log('b = ' + b);
  }
  console.log('finished');
}
block1: {
  console.log('Hello'); // Displays 'Hello'
  break block1;
  console.log('World'); // Will never get here
}
goto block1; // Parse error.
```

Functions

A function is a block with a (possibly empty) parameter list that is normally given a name. A function may use local variables. If you exit the function without a return statement, the value `undefined` is returned.

```
function gcd(segmentA, segmentB) {
  var diff = segmentA - segmentB;
  if (diff == 0)
    return segmentA;
  return diff > 0 ? gcd(segmentB, diff) : gcd(segmentA, -diff);
}
console.log(gcd(60, 40)); // 20
```

```
var mygcd = gcd; // mygcd is a reference to the same function as gcd. Note no argument ()s.
console.log(mygcd(60, 40)); // 20
```

Functions are first class objects and may be assigned to other variables.

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` (that can be implicitly cast to false). 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. (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.)

```
function add7(x, y) {
  if (!y) {
    y = 7;
  }
  console.log(x + y + arguments.length);
};
add7(3); // 11
add7(3, 4); // 9
```

Primitive values (number, boolean, string) 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
console.log(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 full closures by remembering the outer function's local variables even after the outer function has exited.

```
var v = "Top";
var bar, baz;
function foo() {
  var v = "fud";
  bar = function() { console.log(v) };
  baz = function(x) { v = x; };
}
foo();
baz("Fugly");
bar(); // Fugly (not fud) even though foo() has exited.
console.log(v); // Top
```

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 has been assigned to a property name of an object. Unlike many object-oriented languages, there is no distinction between a function definition and a method definition in object-related JavaScript. 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 plain function that initialises an object. When used with the *new* keyword, as is the norm, *this* is set to a newly created blank 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 px() { return this.prefix + "X"; }

function Foo(yz) {
  this.prefix = "a-";
  if (yz > 0) {
    this.pyz = function() { return this.prefix + "Y"; };
  } else {
    this.pyz = function() { return this.prefix + "Z"; };
  }
  this.m1 = px;
  return this;
}

var foo1 = new Foo(1);
var foo2 = new Foo(0);
foo2.prefix = "b-";

console.log("foo1/2 " + foo1.pyz() + foo2.pyz());
// foo1/2 a-Y b-Z

foo1.m3 = px; // Assigns the function itself, not its evaluated result, i.e. not px()
var baz = {"prefix": "c-"};
baz.m4 = px; // No need for a constructor to make an object.

console.log("m1/m3/m4 " + foo1.m1() + foo1.m3() + baz.m4());
// m1/m3/m4 a-X a-X c-X

foo1.m2(); // Throws an exception, because foo1.m2 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 object
console.log(MyObject.staticC); // blue

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

console.log(object.attributeA); // red
console.log(object["attributeB"]); // 1000

console.log(object.staticC); // undefined
object.attributeC = new Date(); // add a new property

delete object.attributeB; // remove a property of object
console.log(object.attributeB); // undefined

delete object; // remove the whole Object (rarely used)
console.log(object.attributeA); // throws an exception
```

The constructor itself is referenced in the object's prototype's *constructor* slot. So,

```
function Foo() {}
// Use of 'new' sets prototype slots (for example,
// x = new Foo() would set x's prototype to Foo.prototype,
// and Foo.prototype has a constructor slot pointing back to Foo).
x = new Foo();
// The 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
// y's prototype is Object.prototype, not
// Foo.prototype, since it was initialised with
// {} instead of new Foo.
// Even though Foo is set to y's constructor slot,
// this is ignored by instanceof - only y's prototype's
// constructor slot is considered.
```

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 `Derived` class inherits from the `Base` class. When `d` is created as `Derived`, 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 `aBaseFunction` *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() { console.log("Base::anOverride()"); };

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

function Derived() {
  this.anOverride = function() { console.log("Derived::anOverride()"); };
}

base = new Base();
Derived.prototype = base; // Must be before new Derived()
Derived.prototype.constructor = Derived; // Required to make `instanceof` work

d = new Derived();    // Copies Derived.prototype to d instance's hidden prototype slot.
d instanceof Derived; // true
d instanceof Base;    // true
```

```
base.aBaseFunction = function() { console.log("Base::aNEWBaseFunction()"); }

d.anOverride();    // Derived::anOverride()
d.aBaseFunction(); // Base::aNEWBaseFunction()
console.log(d.aBaseFunction == Derived.prototype.aBaseFunction); // true

console.log(d.__proto__ == base); // true in Mozilla-based implementations and false in many others.
```

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.m = m2;
bar = new Base();
console.log("bar.m " + bar.m()); // bar.m Two

function Top() { this.m = m3; }
t = new Top();

foo = new Base();
Base.prototype = t;
// No effect on foo, the *reference* to t is copied.
console.log("foo.m " + foo.m()); // foo.m Two

baz = new Base();
console.log("baz.m " + baz.m()); // baz.m Three

t.m = m1; // Does affect baz, and any other derived classes.
console.log("baz.m1 " + baz.m()); // baz.m1 One
```

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

Exception handling

JavaScript includes 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.

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

Native functions and methods

eval (expression)

Evaluates expression string parameter, which can include assignment statements. Variables local to functions can be referenced by the expression. However, the `eval` represents a major security risk, as it allows a bad actor to execute arbitrary code, and so its use is discouraged.[15]

```
(function foo() {
  var x = 7;
  console.log("val " + eval("x + 2"));
})(); // shows val 9.
```

See also

- Comparison of JavaScript-based source code editors
- JavaScript

References

1. JavaScript 1.1 specification (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. "JavaScript Semicolon Insertion: Everything you need to know (http://inimino.org/~inimino/blog/javascript_semicolons)", ~inimino/blog/ (http://inimino.org/~inimino/blog/), Friday, May 28, 2010
4. "Semicolons in JavaScript are optional (http://mislav.uniqpath.com/2010/05/semicolons/)", by Mislav Marohnić, 07 May 2010
5. "Values, Variables, and Literals - MDC" (https://web.archive.org/web/20110629131728/https://developer.mozilla.org/en/JavaScript/Guide/Values%2C_Variables%2C_and_Literals%26revision%3D22#Variables). Mozilla Developer Network. 16 September 2010. Archived from the original (https://developer.mozilla.org/en/JavaScript/Guide/Values,_Variables,_and_Literals&revision=22#Variables) on 29 June 2011. Retrieved 1 February 2020.

6. "JavaScript Scoping and Hoisting (http://www.adequatelygood.com/JavaScript-Scoping-and-Hoisting.html)", Ben Cherry (http://www.adequatelygood.com/about.html), *Adequately Good (http://www.adequatelygood.com/)*, 2010-02-08
7. ECMA-262 5e edition clarified this confusing behavior introducing the notion of *Declarative Environment Record* and *Object Environment Record*. With this formalism, the *global object* is the *Object Environment Record* of the global *Lexical Environment* (the *global scope*).
8. "Template literals" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals). *MDN Web Docs*. Retrieved 2 May 2018.
9. "Comparison Operators - MDC Doc Center" (https://developer.mozilla.org/en/JavaScript/Reference/Operators/Comparison_Operators). Mozilla. 5 August 2010. Retrieved 5 March 2011.
10. "The Elements of JavaScript Style" (http://javascript.crockford.com/style2.html). Douglas Crockford. Retrieved 5 March 2011.
11. "Spread syntax" (https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Spread_operator).
12. "rest parameters" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters).
13. "Ecmascript" (https://web.archive.org/web/20160809104217/https://sebmarkbage.github.io/ecmascript-rest-spread/).
14. ECMA-262, Edition 3, 7.5.3 Future Reserved Words
15. "eval()" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval#Never_use_eval!). *MDN Web Docs*. Retrieved 29 January 2020.

Further reading

- Danny Goodman: *JavaScript Bible*, Wiley, John & Sons, ISBN 0-7645-3342-8.
- David Flanagan, Paula Ferguson: *JavaScript: The Definitive Guide*, O'Reilly & Associates, ISBN 0-596-10199-6.
- Thomas A. Powell, Fritz Schneider: *JavaScript: The Complete Reference*, McGraw-Hill Companies, ISBN 0-07-219127-9.
- Axel Rauschmayer: *Speaking JavaScript: An In-Depth Guide for Programmers*, 460 pages, O'Reilly Media, February 25, 2014, ISBN 978-1449365035. (free online edition (http://speakingjs.com/))
- Emily Vander Veer: *JavaScript For Dummies, 4th Edition*, Wiley, ISBN 0-7645-7659-3.

External links

- A re-introduction to JavaScript - Mozilla Developer Center (https://developer.mozilla.org/en/docs/A_re-introduction_to_JavaScript)
- Comparison Operators in JavaScript (http://www.how-to-code.com/javascript/comparison-operators-in-javascript.html)
- ECMAScript standard references: ECMA-262 (http://www.ecma-international.org/publications/standards/Ecma-262.htm)
- Interactive JavaScript Lessons - example-based (https://web.archive.org/web/20120527095306/http://javalessons.com/cgi-bin/fun/java-tutorials-main.cgi?sub=javascript&code=script)
- JavaScript on About.com: lessons and explanation (http://javascript.about.com/)
- JavaScript Training (https://web.archive.org/web/20150110202627/http://wisentechnologies.com/it-courses/JavaScript-Training-in-Chennai.aspx)

- Mozilla Developer Center Core References for JavaScript versions 1.5 (https://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference), 1.4 (https://web.archive.org/web/20070210000908/http://research.nihonsoft.org/javascript/CoreReferenceJS14/index.html), 1.3 (https://web.archive.org/web/20070210000504/http://research.nihonsoft.org/javascript/ClientReferenceJS13/index.html) and 1.2 (https://web.archive.org/web/20070210000545/http://research.nihonsoft.org/javascript/jsref/index.htm)
- Mozilla JavaScript Language Documentation (https://developer.mozilla.org/en/docs/JavaScript)