Let

ES2015 introduced two important new JavaScript keywords: let and const.

These two keywords provide **Block Scope** variables (and constants) in JavaScript.

Before ES2015, JavaScript had only two types of scope: **Global Scope** and **Function Scope**.

## Global Scope

Variables declared **Globally** (outside any function) have **Global Scope**.

**Global** variables can be accessed from anywhere in a JavaScript program.

## Function Scope

Variables declared **Locally** (inside a function) have **Function Scope**.

**Local** variables can only be accessed from inside the function where they are declared.

Variables declared with the var keyword cannot have **Block Scope**.

Variables declared inside a block **{}** can be accessed from outside the block.

{

    var x = 2;

  }

*// x CAN be used here*

Variables declared with the let keyword can have Block Scope.

Variables declared inside a block **{}** cannot be accessed from outside the block:

{

    let x = 2;

  }

*// x can NOT be used here*

Redeclaring a variable using the var keyword can impose problems.

Redeclaring a variable inside a block will also redeclare the variable outside the block:

var x = 10;

*// Here x is 10*

{

  var x = 2;

*// Here x is 2*

}

*// Here x is 2*

Redeclaring a variable using the let keyword can solve this problem.

Redeclaring a variable inside a block will not redeclare the variable outside the block:

var x = 10;

*// Here x is 10*

{

  let x = 2;

*// Here x is 2*

}

## Loop Scope

Using var in a loop:

var i = 5;

for (var i = 0; i < 10; i++) {

*// some statements*

}

*// Here i is 10*

Using let in a loop:

let i = 5;

for (let i = 0; i < 10; i++) {

*// some statements*

}

*// Here i is 5*

## Function Scope

Variables declared with var and let are quite similar when declared inside a function.

function myFunction() {

    var carName = "Volvo";   *// Function Scope*

  }

  function myFunction() {

    let carName = "Volvo";   *// Function Scope*

  }

## Global Scope

Variables declared with var and let are quite similar when declared outside a block.

var x = 2;       *// Global scope*

  let x = 2;       *// Global scope*

## Global Variables in HTML

In HTML, the global scope is the window object.

Global variables defined with the var keyword belong to the window object:

var carName = "Volvo";

*// code here can use window.carName*

Global variables defined with the let keyword do not belong to the window object:

let carName = "Volvo";

*// code here cannot use window.carName*

## Redeclaring

Redeclaring a JavaScript variable with var is allowed anywhere in a program:

var x = 2;

*// Now x is 2*

var x = 3;

*// Now x is 3*

Redeclaring a var variable with let, in the same scope, or in the same block, is not allowed:

var x = 2;       *// Allowed*

let x = 3;       *// Not allowed*

{

  var x = 4;   *// Allowed*

  let x = 5   *// Not allowed*

}

Redeclaring a let variable with let, in the same scope, or in the same block, is not allowed:

let x = 2;       *// Allowed*

let x = 3;       *// Not allowed*

{

  let x = 4;   *// Allowed*

  let x = 5;   *// Not allowed*

}

Redeclaring a let variable with var, in the same scope, or in the same block, is not allowed:

let x = 2;       *// Allowed*

var x = 3;       *// Not allowed*

{

  let x = 4;   *// Allowed*

  var x = 5;   *// Not allowed*

}

Redeclaring a variable with let, in another scope, or in another block, is allowed:

let x = 2;       *// Allowed*

{

  let x = 3;   *// Allowed*

}

{

  let x = 4;   *// Allowed*

}

## Hoisting

Variables defined with var are **hoisted** to the top with undefined and can be initialized at any time

carName = "Volvo";

alert(carName);

var carName;

Variables defined with let are hoisted to the top of the block, but not initialized.

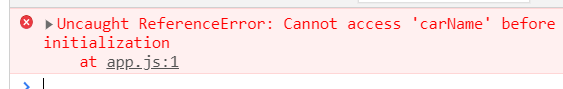
Meaning: The block of code is aware of the variable, but it cannot be used until it has been declared.

Using a let variable before it is declared will result in a ReferenceError.

The variable is in a "temporal dead zone" from the start of the block until it is declared:

carName = "Volvo";

let carName;



# Const

# Variables defined with const behave like let variables, except they cannot be reassigned:

const PI = 3.141592653589793;

PI = 3.14;      *// This will give an error*

PI = PI + 10;   *// This will also give an error*

# 

## Block Scope

Declaring a variable with const is similar to let when it comes to **Block Scope**.

var x = 10;

*// Here x is 10*

{

  const x = 2;

*// Here x is 2*

}

*// Here x is 10*

## Assigned when Declared

JavaScript const variables must be assigned a value when they are declared:

const PI;

PI = 3.14159265359;

# 

## Not Real Constants

we cannot change constant primitive values, but we can change the properties of constant objects.

## Primitive Values

If we assign a primitive value to a constant, we cannot change the primitive value:

const PI = 3.141592653589793;

PI = 3.14;



## Constant Objects can Change

You can change the properties of a constant object:

*// You can create a const object:*

const car = {type:"Fiat", model:"500", color:"white"};

*// You can change a property:*

car.color = "red";

*// You can add a property:*

car.owner = "Johnson";

But you can NOT reassign a constant object:

const car = {type:"Fiat", model:"500", color:"white"};

car = {type:"Volvo", model:"EX60", color:"red"}; *//Error:Uncaught TypeError: Assignment to constant variable*



## Constant Arrays can Change

You can change the elements of a constant array:

*// You can create a constant array:*

const cars = ["Saab", "Volvo", "BMW"];

*// You can change an element:*

cars[0] = "Toyota";

*// You can add an element:*

cars.push("Audi");

But you can NOT reassign a constant array:

const cars = ["Saab", "Volvo", "BMW"];

cars = ["Toyota", "Volvo", "Audi"];    *// ERROR*



## Redeclaring

Redeclaring or reassigning an existing var or let variable to const, in the same scope, or in the same block, is **not allowed**:

var x = 2;         *// Allowed*

const x = 2;       *// Not allowed*

{

  let x = 2;     *// Allowed*

  const x = 2;   *// Not allowed*

}



Redeclaring or reassigning an existing const variable, in the same scope, or in the same block, is not allowed:

const x = 2;       *// Allowed*

const x = 3;       *// Not allowed*

x = 3;             *// Not allowed*

var x = 3;         *// Not allowed*

let x = 3;         *// Not allowed*

{

  const x = 2;   *// Allowed*

  const x = 3;   *// Not allowed*

  x = 3;         *// Not allowed*

  var x = 3;     *// Not allowed*

  let x = 3;     *// Not allowed*

}

Redeclaring a variable with const, in another scope, or in another block, is allowed:

const x = 2;       *// Allowed*

{

  const x = 3;   *// Allowed*

}

{

  const x = 4;   *// Allowed*

}

## Exponentiation

The **exponentiation** operator (\*\*) raises the first operand to the power of the second operand.

var x = 5;

var z = x \*\* 2;

x \*\* y produces the same result as Math.pow(x,y):

# JavaScript Functions

A JavaScript function is a block of code designed to perform a particular task.

When JavaScript reaches a return statement, the function will stop executing.

## Why Functions?

You can reuse code: Define the code once, and use it many times.

You can use the same code many times with different arguments, to produce different results.

## The () Operator Invokes the Function

## Functions Used as Variable Values

Functions can be used the same way as you use variables, in all types of formulas, assignments, and calculations.

## Local Variables

Variables declared within a JavaScript function, become **LOCAL** to the function.

Local variables can only be accessed from within the function.

*// code here can NOT use carName*

function myFunction() {

    var carName = "Volvo";

*// code here CAN use carName*

  }

*// code here can NOT use carName*

# JavaScript Objects

JavaScript objects are containers for **named values**called properties or methods.

## The ****this**** Keyword

In a function definition, this refers to the "owner" of the function.

## Do Not Declare Strings, Numbers, and Booleans as Objects!

When a JavaScript variable is declared with the keyword "new", the variable is created as an object:

var x = new String();        *// Declares x as a String object*

var y = new Number();        *// Declares y as a Number object*

var z = new Boolean();       *// Declares z as a Boolean object*

Avoid String, Number, and Boolean objects. They complicate your code and slow down execution speed.

# JavaScript Strings

## Escape Character

Because strings must be written within quotes, JavaScript will misunderstand this string:

var x = "We are the so-called "Vikings" from the north.";

use the **backslash escape character**.

var x = "We are the so-called \"Vikings\" from the north.";

## Strings Can be Objects

Normally, JavaScript strings are primitive values, created from literals:

var firstName = "John";

But strings can also be defined as objects with the keyword new:

var firstName = new String("John");

var x = "John";

var y = new String("John");

*// typeof x will return string*

*// typeof y will return object*

var x = "John";

var y = new String("John");

*// (x == y) is true because x and y have equal values*

var x = "John";

var y = new String("John");

*// (x === y) is false because x and y have different types (string and object)*

Or even worse. Objects cannot be compared:

var x = new String("John");

var y = new String("John");

*// (x == y) is false because x and y are different objects*

Comparing two JavaScript objects will **always** return false.

# JavaScript String Methods

## Finding a String in a String

The indexOf() method returns the index of (the position of) the first occurrence of a specified text in a string:

var str = "Please locate where 'locate' occurs!";

var pos = str.indexOf("locate"); *//output: 7*

The lastIndexOf() method returns the index of the **last** occurrence of a specified text in a string:

var pos = str.lastIndexOf("locate"); *//output: 21*

Both indexOf(), and lastIndexOf() return -1 if the text is not found.

Both methods accept a second parameter as the starting position for the search:

var pos = str.indexOf("locate", 15); *//output: 21*

The lastIndexOf() methods searches backwards (from the end to the beginning), meaning: if the second parameter is 15, the search starts at position 15, and searches to the beginning of the string.

var pos = str.lastIndexOf("locate", 15); *//output: 7*

## Searching for a String in a String

The search() method searches a string for a specified value and returns the position of the match:

var pos = str.search("locate"); *//output: 7*

The two methods are **NOT** equal. These are the differences:

* The search() method cannot take a second start position argument.
* The indexOf() method cannot take powerful search values (regular expressions).

## Extracting String Parts

There are 3 methods for extracting a part of a string:

slice(start, end)

substring(start, end)

substr(start, length)

slice() extracts a part of a string and returns the extracted part in **a new string**.

The method takes 2 parameters: the start position, and the end position (end not included).

var str = "Apple, Banana, Kiwi";

var res = str.slice(7, 13); *//output: Banana*

If a parameter is negative, the position is counted from the end of the string.

var str = "Apple, Banana, Kiwi";

var res = str.slice(-12, -6); *//output: Banana*

If you omit the second parameter, the method will slice out the rest of the string:

var str = "Apple, Banana, Kiwi";

var res = str.slice(7); *//output: Banana, Kiwi*

or, counting from the end:

var str = "Apple, Banana, Kiwi";

var res = str.slice(-12); *//output: Banana, Kiwi*

## The substring() Method

substring() is similar to slice().

The difference is that substring() cannot accept negative indexes.

var str = "Apple, Banana, Kiwi";

var res = str.substring(7,13); *//Banana, Kiwi*

If you omit the second parameter, substring() will slice out the rest of the string.

var str = "Apple, Banana, Kiwi";

var res = str.substring(7); *//Banana, Kiwi*

## The substr() Method

substr() is similar to slice().

The difference is that the second parameter specifies the **length** of the extracted part.

var str = "Apple, Banana, Kiwi";

var res = str.substr(7, 6); *//Banana*

If you omit the second parameter, substr() will slice out the rest of the string.

var str = "Apple, Banana, Kiwi";

var res = str.substr(7); *//Banana, Kiwi*

If the first parameter is negative, the position counts from the end of the string.

var str = "Apple, Banana, Kiwi";

var res = str.substr(-4); *//Kiwi*

## Replacing String Content

str = "Please visit Microsoft!";

var n = str.replace("Microsoft", "W3Schools"); *//Please visit W3Schools!*

The replace() method does not change the string it is called on. It returns a new string.

By default, the replace() method replaces **only the first** match:

By default, the replace() method is case sensitive. Writing MICROSOFT (with upper-case) will not work:

str = "Please visit Microsoft!";

var n = str.replace("MICROSOFT", "W3Schools"); *//Please visit Microsoft!*

To replace case insensitive, use a **regular expression** with an /i flag (insensitive):

str = "Please visit Microsoft!";

var n = str.replace(/MICROSOFT/i, "W3Schools"); *//Please visit W3Schools!*

Note that regular expressions are written without quotes.

To replace all matches, use a **regular expression** with a /g flag (global match):

str = "Please visit Microsoft and Microsoft!";

var n = str.replace(/Microsoft/g, "W3Schools");

## The concat() Method

concat() joins two or more strings:

var text1 = "Hello";

var text2 = "World";

var text3 = text1.concat(" ", text2); *//Hello World*

## String.trim()

The trim() method removes whitespace from both sides of a string:

## JavaScript String Padding

ECMAScript 2017 added two String methods: padStart and padEnd to support padding at the beginning and at the end of a string.

let str = "5";

str = str.**padStart**(4,0);*// result is 0005*

let str = "5";

str = str.**padEnd**(4,0);*// result is 5000*

## Extracting String Characters

There are 3 methods for extracting string characters:

* charAt(position)
* charCodeAt(position)
* Property access [ ]

## The charAt() Method

The charAt() method returns the character at a specified index (position) in a string:

var str = "HELLO WORLD";

console.log(str.charAt(0)) *// returns H*

## The charCodeAt() Method

The charCodeAt() method returns the unicode of the character at a specified index in a string:

The method returns a UTF-16 code (an integer between 0 and 65535).

var str = "HELLO WORLD";

console.log(str.charCodeAt(0)) *// returns 72*

## Property Access

ECMAScript 5 (2009) allows property access [ ] on strings:

var str = "HELLO WORLD";

str[0];                   *// returns H*

Property access might be a little **unpredictable:**

* It does not work in Internet Explorer 7 or earlier
* It makes strings look like arrays (but they are not)
* If no character is found, [ ] returns undefined, while charAt() returns an empty string.
* It is read only. str[0] = "A" gives no error (but does not work!)

var str = "HELLO WORLD";

str[0] = "A";             *// Gives no error, but does not work*

console.log(str[0]);                   *// returns H*

## Converting a String to an Array

A string can be converted to an array with the split() method:

var txt = "a,b,c,d,e";   *// String*

txt.split(",");          *// Split on commas*

txt.split(" ");          *// Split on spaces*

txt.split("|");          *// Split on pipe*

# JavaScript Numbers

JavaScript has only one type of number. Numbers can be written with or without decimals.

Extra large or extra small numbers can be written with scientific (exponent) notation:

var x = 123e5;    *// 12300000*

var y = 123e-5;   *// 0.00123*

## Precision

Integers (numbers without a period or exponent notation) are accurate up to **15 digits**.

var x = 999999999999999;   *// x will be 999999999999999*

var y = 9999999999999999;  *// y will be 10000000000000000*

The maximum number of decimals is 17, but floating point arithmetic is not always 100% accurate:

var x = 0.2 + 0.1;         *// x will be 0.30000000000000004*

var x = 10;

var y = 20;

var z = "30";

var result = x + y + z; *//3030*

JavaScript will try to convert strings to numbers in all numeric operations:

var x = "100";

var y = "10";

var z = x / y;       *// z will be 10*

This will also work:

var x = "100";

var y = "10";

var z = x \* y;       *// z will be 1000*

This will work as:

var x = "100";

var y = "10";

var z = x - y;       *// z will be 90*

JavaScript uses the + operator to concatenate the strings.

## NaN - Not a Number

NaN is a JavaScript reserved word indicating that a number is not a legal number.

Trying to do arithmetic with a non-numeric string will result in NaN (Not a Number):

var x = 100 / "Apple";  *// x will be NaN (Not a Number)*

You can use the global JavaScript function isNaN() to find out if a value is a number:

var x = 100 / "Apple";

isNaN(x);               *// returns true because x is Not a Number*

Watch out for NaN. If you use NaN in a mathematical operation, the result will also be NaN:

var x = NaN;

var y = 5;

var z = x + y;         *// z will be NaN*

Or the result might be a concatenation:

var x = NaN;

var y = "5";

var z = x + y;         *// z will be NaN5*

NaN is a number: typeof NaN returns number:

typeof NaN *// number*

Infinity (or -Infinity) is the value JavaScript will return if you calculate a number outside the largest possible number.

var myNumber = 2;

while (myNumber != Infinity) {   *// Execute until Infinity*

  myNumber = myNumber \* myNumber;

  console.log(myNumber)

}

Division by 0 (zero) also generates Infinity:

var x =  2 / 0;       *// x will be Infinity*

var y = -2 / 0;       *// y will be -Infinity*

Infinity is a number: typeof Infinity returns number.

By default, JavaScript displays numbers as **base 10** decimals.

But you can use the toString() method to output numbers from **base 2** to **base 36**.

Hexadecimal is **base 16**. Decimal is **base 10**. Octal is **base 8**. Binary is **base 2**.

var myNumber = 32;

myNumber.toString(10);  *// returns 32*

myNumber.toString(32);  *// returns 10*

myNumber.toString(16);  *// returns 20*

myNumber.toString(8);   *// returns 40*

myNumber.toString(2);   *// returns 100000*

## Numbers Can be Objects

Normally JavaScript numbers are primitive values created from literals:

var x = 123;

But numbers can also be defined as objects with the keyword new:

var y = new Number(123);

var x = 123;*// typeof x returns number*

var y = new Number(123);*// typeof y returns object*

Do not create Number objects. It slows down execution speed.  
The new keyword complicates the code. This can produce some unexpected results:

var x = 500;            *//* *number*

var y = new Number(500); *//* *object*

*// (x == y) is true because x and y have equal values*

*// (x === y) is false because x and y have different types*

Or even worse. Objects cannot be compared:

var x = new Number(500);

var y = new Number(500);

*// (x == y) is false because objects cannot be compared*

# JavaScript Number Methods

## The toString() Method

The toString() method returns a number as a string.

var x = 123;

x.toString();            *// returns 123 from variable x*

(123).toString();        *// returns 123 from literal 123*

(100 + 23).toString();   *// returns 123 from expression 100 + 23*

## The toExponential() Method

toExponential() returns a string, with a number rounded and written using exponential notation.

A parameter defines the number of characters behind the decimal point:

var x = 9.656;

x.toExponential(); *//If you don't specify it, JavaScript will not round the number.*

x.toExponential(2);     *// returns 9.66e+0*

x.toExponential(4);     *// returns 9.6560e+0*

x.toExponential(6);     *// returns 9.656000e+0*

## The toFixed() Method

toFixed() returns a string, with the number written with a specified **number of decimals**:

var x = 9.656;

x.toFixed();           *// returns 10*

x.toFixed(0);           *// returns 10*

x.toFixed(2);           *// returns 9.66*

x.toFixed(4);           *// returns 9.6560*

x.toFixed(6);           *// returns 9.656000*

toFixed(2) is perfect for working with money.

## The toPrecision() Method

toPrecision() returns a string, with a number written with a **specified length**:

var x = 9.656;

 x.toPrecision()    *//Error:argument must be between 1 and 100*

x.toPrecision(0);   *//Error:argument must be between 1 and 100*

x.toPrecision(2);       *// returns 9.7*

x.toPrecision(4);       *// returns 9.656*

x.toPrecision(6);       *// returns 9.65600*

## The valueOf() Method

valueOf() returns a number as a number.

var x = 123;

x.valueOf();            *// returns 123 from variable x*

(123).valueOf();        *// returns 123 from literal 123*

(100 + 23).valueOf();   *// returns 123 from expression 100 + 23*

The valueOf() method is used internally in JavaScript to convert Number objects to primitive values.

There is no reason to use it in your code.

## Converting Variables to Numbers

There are 3 JavaScript methods that can be used to convert variables to numbers:

* The Number() method
* The parseInt() method
* The parseFloat() method

These methods are not **number** methods, but **global** JavaScript methods.

|  |  |
| --- | --- |
| **Method** | **Description** |
| Number() | Returns a number, converted from its argument. |
| parseFloat() | Parses its argument and returns a floating point number |
| parseInt() | Parses its argument and returns an integer |

Number(true);          *// returns 1*

Number(false);         *// returns 0*

Number("10");          *// returns 10*

Number("  10");        *// returns 10*

Number("10  ");        *// returns 10*

Number(" 10  ");       *// returns 10*

Number("10.33");       *// returns 10.33*

Number("10,33");       *// returns NaN*

Number("10 33");       *// returns NaN*

Number("John");        *// returns NaN*

If the number cannot be converted, NaN (Not a Number) is returned.

## The Number() Method Used on Dates

Number() can also convert a date to a number:

Number(new Date("2017-09-30"));

The Number() method above returns the number of milliseconds since 1.1.1970.

## The parseInt() Method

parseInt() parses a string and returns a whole number. Spaces are allowed. Only the first number is returned:

parseInt("10");         *// returns 10*

parseInt("10.33");      *// returns 10*

parseInt("10 20 30");   *// returns 10*

parseInt("10 years");   *// returns 10*

parseInt("years 10");   *// returns NaN*

If the number cannot be converted, NaN (Not a Number) is returned.

## The parseFloat() Method

parseFloat() parses a string and returns a number. Spaces are allowed. Only the first number is returned:

parseFloat("10");        *// returns 10*

parseFloat("10.33");     *// returns 10.33*

parseFloat("10 20 30");  *// returns 10*

parseFloat("10 years");  *// returns 10*

parseFloat("years 10");  *// returns NaN*

## Number Properties

|  |  |
| --- | --- |
| **Property** | **Description** |
| MAX\_VALUE | Returns the largest number possible in JavaScript |
| MIN\_VALUE | Returns the smallest number possible in JavaScript |
| POSITIVE\_INFINITY | Represents infinity (returned on overflow) |
| NEGATIVE\_INFINITY | Represents negative infinity (returned on overflow) |
| NaN | Represents a "Not-a-Number" value |

## JavaScript MIN\_VALUE and MAX\_VALUE

MAX\_VALUE returns the largest possible number in JavaScript.

var x = Number.MAX\_VALUE;

var y = Number.MIN\_VALUE;

## JavaScript POSITIVE\_INFINITY and NEGATIVE\_INFINITY

## POSITIVE\_INFINITY is returned on overflow:

var x = 1 / 0; *//Infinity*

var x1 = Number.POSITIVE\_INFINITY; *//Infinity*

var y = -1 / 0; *//-Infinity*

var y1 = Number.NEGATIVE\_INFINITY; *//-Infinity*

## JavaScript NaN - Not a Number

var x = Number.NaN; *//NaN*

NaN is a JavaScript reserved word indicating that a number is not a legal number.

## Number Properties Cannot be Used on Variables

Number properties belongs to the JavaScript's number object wrapper called **Number**.

var x = 6;

var y = x.MAX\_VALUE;    *// y becomes undefined*

# JavaScript Arrays

JavaScript arrays are used to store multiple values in a single variable.

## Using the JavaScript Keyword new

var cars = new Array("Saab", "Volvo", "BMW");

## Arrays are Objects

Arrays are a special type of objects. The typeof operator in JavaScript returns "object" for arrays.

## Array Properties and Methods

The real strength of JavaScript arrays are the built-in array properties and methods:

var y = cars.sort();   *// The sort() method sorts arrays*

## The Difference Between Arrays and Objects

In JavaScript, **arrays** use **numbered indexes**.

In JavaScript, **objects** use **named indexes**.

## Avoid new Array()

There is no need to use the JavaScript's built-in array constructor new Array().

**Use [] instead.**

var points = new Array();     *// Bad*

var points2 = [];              *// Good*

The new keyword only complicates the code. It can also produce some unexpected results:

var points = new Array(40, 100);  *// Creates an array with two elements (40 and 100)*

var points2 = new Array(40);  *// Creates an array with 40 undefined elements !!!!!*

## How to Recognize an Array

The problem is that the JavaScript operator typeof returns "object":

To solve this problem ECMAScript 5 defines a new method Array.isArray():

var fruits = ["Banana", "Orange", "Apple", "Mango"];

Array.isArray(fruits);   *// returns true*

To solve this problem you can create your own isArray() function:

var fruits =["Banana", "Orange", "Apple", "Mango"];

function isArray(myArray) {

    return myArray.constructor.toString().indexOf("Array") > -1;

}

The instanceof operator returns **true** if an object is created by a given constructor:

var fruits =["Banana", "Orange", "Apple", "Mango", "Array"];

var isArray = fruits instanceof Array;   *// returns true*

# JavaScript Array Methods

## Converting Arrays to Strings

The JavaScript method toString() converts an array to a string of (comma separated) array values.

var fruits =["Banana", "Orange", "Apple", "Mango", "Array"];

var isArray = fruits.toString();   *// returns Banana,Orange,Apple,Mango,Array*

The join() method also joins all array elements into a string.

It behaves just like toString(), but in addition you can specify the separator:

var fruits =["Banana", "Orange", "Apple", "Mango", "Array"];

var isArray = fruits.join(" \* ");   *// returns Banana \* Orange \* Apple \* Mango \* Array*

## Popping and Pushing

var fruits = ["Banana", "Orange", "Apple", "Mango"];

var fruit = fruits.pop();     *//return"Mango", Removes the last element ("Mango") from fruits*

# The pop() method returns the value that was "popped out":

# The push() method adds a new element to an array (at the end):

var fruits = ["Banana", "Orange", "Apple", "Mango"];

var fruit = fruits.push("Kiwi"); *// return 5*

The push() method returns the new array length:

## Shifting Elements

The shift() method removes the first array element and "shifts" all other elements to a lower index.

var fruits = ["Banana", "Orange", "Apple", "Mango"];

var fruit = fruits.shift();   *//return "Banana", Removes the first element "Banana" from fruits*

The shift() method returns the **string that was "shifted out":**

The unshift() method adds a new element to an array (at the beginning), and "unshifts" older elements:

var fruits = ["Banana", "Orange", "Apple", "Mango"];

var fruit = fruits.unshift("Lemon");    *// return 5, Adds a new element "Lemon" to fruits*

The unshift() method returns the **new array length**.

## Deleting Elements

Since JavaScript arrays are objects, elements can be deleted by using the JavaScript operator delete:

var fruits = ["Banana", "Orange", "Apple", "Mango"];

delete fruits[0]; *//return true   // Changes the first element in fruits to undefined*

console.log(fruits) *// [empty(undefined), "Orange", "Apple", "Mango"]*

Using **delete** may leave undefined holes in the array. Use pop() or shift() instead.

## Splicing an Array

The splice() method can be used to add new items to an array:

var fruits = ["Banana", "Orange", "Apple", "Mango"];

console.log(fruits.splice(2, 1, "Lemon", "Kiwi")) *//return ["Apple"]*

console.log(fruits) *//return ["Banana", "Orange", "Lemon", "Kiwi", "Mango"]*

The first parameter (2) defines the position **where** new elements should be **added** (spliced in).

The second parameter (1) defines **how many** elements should be **removed**.

The rest of the parameters ("Lemon" , "Kiwi") define the new elements to be **added**.

The splice() method returns an array with the deleted items:

## Using splice() to Remove Elements

With clever parameter setting, you can use splice() to remove elements without leaving "holes" in the array:

var fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.splice(0, 1);        *// Removes the first element of fruits*

The first parameter (0) defines the position where new elements should be **added** (spliced in).

The second parameter (1) defines **how many** elements should be **removed**.

The rest of the parameters are omitted. No new elements will be added.

## Merging (Concatenating) Arrays

The concat() method creates a new array by merging (concatenating) existing arrays:

var myGirls = ["Cecilie", "Lone"];

var myBoys = ["Emil", "Tobias", "Linus"];

var myChildren = myGirls.concat(myBoys);   *// Concatenates (joins) myGirls and myBoys*

The concat() method does not change the existing arrays. It always returns a new array.

The concat() method can take any number of array arguments:

var arr1 = ["Cecilie", "Lone"];

var arr2 = ["Emil", "Tobias", "Linus"];

var arr3 = ["Robin", "Morgan"];

var arr4 = ["Robin1", "Morgan2"];

var arr5 = ["Robin3", "Morgan4"];

var arr6 = ["Robin5", "Morgan6"];

var myChildren = arr1.concat(arr2, arr3, arr4,arr5, arr6);   *// Concatenates arr1 with arr2 and arr3 ...*

The concat() method can also take strings as arguments:

var arr1 = ["Emil", "Tobias", "Linus"];

var myChildren = arr1.concat("Peter"); *//output: ["Emil", "Tobias", "Linus", "Peter"]*

var myChildren2 = arr1.concat("Peter","Mark");  *//output: ["Emil", "Tobias", "Linus", "Peter", "Mark"]*

## Slicing an Array

The slice() method slices out a piece of an array into a new array.

var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];

var citrus = fruits.slice(1);

console.log(fruits) *//["Banana", "Orange", "Lemon", "Apple", "Mango"]*

console.log(citrus) *//["Orange", "Lemon", "Apple", "Mango"]*

The slice() method creates a new array. It does not remove any elements from the source array.

The method then selects elements from the start argument, and up to (but not including) the end argument.

var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];

var citrus = fruits.slice(1, 3);*//["Orange", "Lemon"]*

If the end argument is omitted, like in the first examples, the slice() method slices out the rest of the array.

var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];

var citrus = fruits.slice(2);*//["Lemon", "Apple", "Mango"]*

# JavaScript Sorting Arrays

The sort() method sorts an array alphabetically:

var fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.sort();        *// Sorts the elements of fruits*

## Reversing an Array

The reverse() method reverses the elements in an array.

var fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.reverse();     *// reverse the order of the elements*

## Numeric Sort

By default, the sort() function sorts values as **strings**.

However, if numbers are sorted as strings, "**2**5" is bigger than "**1**00", because "2" is bigger than "1".

You can fix this by providing a **compare function**:

var points = [40, 100, 1, 5, 25, 10];

points.sort(function(a, b){return a - b}); *//[1, 5, 10, 25, 40, 100]*

Use the same trick to sort an array descending:

var points = [40, 100, 1, 5, 25, 10];

points.sort(function(a, b){return b - a});

## The Compare Function

The compare function should return a negative, zero, or positive value, depending on the arguments: function(a, b){return a - b}

When the sort() function compares two values, it sends the values to the compare function, and sorts the values according to the returned (negative, zero, positive) value.

If the result is negative a is sorted before b.

If the result is positive b is sorted before a.

If the result is 0 no changes are done with the sort order of the two values.

## Sorting an Array in Random Order

var points = [40, 100, 1, 5, 25, 10];

points.sort(function(a, b){return 0.5 - Math.random()});

## The Fisher Yates Method

The above example, array.sort(), is not accurate, it will favor some numbers over the others.

The most popular correct method, is called the Fisher Yates shuffle, and was introduced in data science as early as 1938!

var points = [40, 100, 1, 5, 25, 10];

for (i = points.length -1; i > 0; i--) {

  j = Math.floor(Math.random() \* i)

  k = points[i]

  points[i] = points[j]

  points[j] = k

}

## ­Find the Highest (or Lowest) Array Value

here are no built-in functions for finding the max or min value in an array.

var points = [40, 100, 1, 5, 25, 10];

points.sort(function(a, b){return a - b});

*// now points[0] contains the lowest value*

*// and points[points.length-1] contains the highest value*

Sorting a whole array is a very inefficient method if you only want to find the highest (or lowest) value.

## Using Math.max() on an Array

ou can use Math.max.apply to find the highest number in an array:

var max = Math.max(1, 2, 3); *//3*

var max1 = Math.max.call(null,1, 2, 3);*//3*

var max2 = Math.max.apply(null, [1, 2, 3]);*//3*

var max3 = Math.max.bind(null, 1, 2, 3); *// ƒ max() { [native code] }*

You can use Math.min.apply to find the lowest number in an array:

var min = Math.min(1, 2, 3); *//1*

var min1 = Math.min.call(null,1, 2, 3);*//1*

var min2 = Math.min.apply(null, [1, 2, 3]);*//1*

var min3 = Math.min.bind(null, 1, 2, 3); *// ƒ min() { [native code] }*

## My Min / Max JavaScript Methods

This function loops through an array comparing each value with the highest value found:

function myArrayMax(arr) {

    var len = arr.length;

    var max = -Infinity;

    while (len--) {

      if (arr[len] > max) {

        max = arr[len];

      }

    }

    return max;

  }

  console.log(myArrayMax([40, 100, 1, 5, 25, 10])) *//100*

This function loops through an array comparing each value with the lowest value found:

function myArrayMin(arr) {

    var len = arr.length;

    var min = Infinity;

    while (len--) {

      if (arr[len] < min) {

        min = arr[len];

      }

    }

    return min;

  }

  console.log(myArrayMin([40, 100, 1, 5, 25, 10])) *//1*

## Sorting Object Arrays

Even if objects have properties of different data types, the sort() method can be used to sort the array.

var cars = [

    {type:"Volvo", year:2016},

    {type:"Saab", year:2001},

    {type:"BMW", year:2010}

  ];

  var sortedcars = cars.sort(function(a, b){return a.year - b.year});

Comparing string properties is a little more complex:

var sortedcars = cars.sort(function(a, b){

    var x = a.type.toLowerCase();

    var y = b.type.toLowerCase();

    if (x < y) {return -1;}

    if (x > y) {return 1;}

    return 0;

  });

# JavaScript Array Iteration Methods

var numbers = [45, 4, 9, 16, 25];

numbers.forEach((value, index, array)=>{console.log(value, index, array)});

Note that the function takes 3 arguments:

* The item value
* The item index
* The array itself

## Array.map()

The map() method creates a new array by performing a function on each array element.

The map() method does not execute the function for array elements without values.

The map() method does not change the original array.

var numbers1 = [45, 4, 9, 16, 25];

var numbers2 = numbers1.map((value, index, array)=> {

    return value \* 2;

  });

console.log(numbers1, numbers2); *//[45, 4, 9, 16, 25] (5) [90, 8, 18, 32, 50]*

## Array.filter()

The filter() method creates a new array with array elements that passes a test.

var numbers = [45, 4, 9, 16, 25];

var over18 = numbers.filter((value, index, array) => {

    return value > 18;

  });

console.log(numbers, over18); *//[45, 4, 9, 16, 25] , [45, 25]*

## Array.reduce()

The reduce() method runs a function on each array element to produce (reduce it to) a single value.

The reduce() method works from left-to-right in the array.

Note that the function takes 4 arguments:

* The total (the initial value / previously returned value)
* The item value
* The item index
* The array itself

The reduce() method does not reduce the original array.

var numbers1 = [45, 4, 9, 16, 25];

var sum = numbers1.reduce((total, value, index, array) =>{

    return total + value;

  });

console.log(numbers1, sum); *//[45, 4, 9, 16, 25] 99*

The reduce() method can accept an initial value:

var numbers1 = [45, 4, 9, 16, 25];

var sum = numbers1.reduce((total, value, index, array) =>{

    return total + value;

  },**100**);

console.log(numbers1, sum); *//[45, 4, 9, 16, 25] 199*

## Array.reduceRight()

The reduceRight() works from right-to-left in the array. See also reduce().

var numbers1 = [45, 4, 9, 16, 25];

var sum = numbers1.reduceRight((total, value, index, array) =>{

    return total + value;

  },100);

console.log(numbers1, sum); *//[45, 4, 9, 16, 25] 199*

## Array.every()

The every() method check if all array values pass a test.

Note that the function takes 3 arguments:

* The item value
* The item index
* The array itself
* var numbers1 = [45, 4, 9, 16, 25];
* var allOver18 = numbers1.every((value, index, array) =>{
* return value > 18;
* });
* console.log(numbers1, allOver18); *//[45, 4, 9, 16, 25] false*

## Array.some()

The some() method check if some array values pass a test.

var numbers1 = [45, 4, 9, 16, 25];

var allOver18 = numbers1.some((value, index, array) =>{

    return value > 18;

  });

 console.log(numbers1, allOver18); *//[45, 4, 9, 16, 25] true*

## Array.indexOf()

*array*.indexOf(item, start)

|  |  |
| --- | --- |
| start | Optional. Where to start the search. Negative values will start at the given position counting from the end, and search to the end. |

var fruits = ["Apple", "Orange", "Apple", "Mango"];

var a = fruits.indexOf("Apple",2); *//2*

The indexOf() method searches an array for an element value and returns its position.

var fruits = ["Apple", "Orange", "Apple", "Mango"];

var a = fruits.indexOf("Orange"); //1

Array.indexOf() returns -1 if the item is not found.

If the item is present more than once, it returns the position of the first occurrence.

## Array.lastIndexOf()

Array.lastIndexOf() is the same as Array.indexOf(), but returns the position of the last occurrence of the specified element.

var fruits = ["Apple", "Orange", "Apple", "Mango"];

var a = fruits.lastIndexOf("Apple"); *//2*

## Array.find()

The find() method returns the value of the first array element that passes a test function.

var numbers = [4, 9, 16, 25, 29];

var first = numbers.find((value, index, array) => value > 18);

console.log(first)*//25*

## Array.findIndex()

The findIndex() method returns the index of the first array element that passes a test function.

var numbers = [4, 9, 16, 25, 29];

var first = numbers.findIndex((value, index, array) => value > 18);

console.log(first)*//3*

# JavaScript Date Objects

## JavaScript Date Output

By default, JavaScript will use the browser's time zone and display a date as a full text string:

## Creating Date Objects

Date objects are created with the new Date() constructor.

There are **4 ways** to create a new date object:

new Date()  
new Date(year, month, day, hours, minutes, seconds, milliseconds)  
new Date(milliseconds)  
new Date(date string)

Date objects are static. The computer time is ticking, but date objects are not.

## new Date(year, month, ...)

new Date(year, month, ...) creates a new date object with a **specified date and time**.

7 numbers specify year, month, day, hour, minute, second, and millisecond (in that order):

var d = new Date(2018, 11, 24, 10, 33, 30, 0);

console.log(d)*//  Mon Dec 24 2018 10:33:30 GMT+0530 (India Standard Time)*

**Note:** JavaScript counts months from 0 to 11.

January is 0. December is 11.

var date7Number = new Date(2018, 11, 24, 10, 33, 30, 0);

var date6Number = new Date(2018, 11, 24, 10, 33, 30);

var date5Number = new Date(2018, 11, 24, 10, 33);

var date4Number = new Date(2018, 11, 24, 10);

var date3Number = new Date(2018, 11, 24);

var date2Number = new Date(2018, 11);

var date1Number = new Date(2018);

You cannot omit month. If you supply only one parameter it will be treated as milliseconds.

## Previous Century

One and two digit years will be interpreted as 19xx:

var date1Number = new Date(99, 11, 24); *//Fri Dec 24 1999 00:00:00 GMT+0530 (India Standard Time)*

var d = new Date(9, 11, 24); *//Fri Dec 24 1909 00:00:00 GMT+0530 (India Standard Time)*

## new Date(dateString)

new Date(dateString) creates a new date object from a **date string**:

var d = new Date("October 13, 2014 11:13:00");*//Mon Oct 13 2014 11:13:00 GMT+0530 (India Standard Time)*

## JavaScript Stores Dates as Milliseconds

JavaScript stores dates as number of milliseconds since **January 01, 1970**, 00:00:00 UTC (Universal Time Coordinated).

## new Date(milliseconds)

new Date(milliseconds) creates a new date object as**zero time plus milliseconds**:

01 January 1970 **plus** 100 000 000 000 milliseconds is approximately 03 March 1973:

var d = new Date(100000000000); *//Sat Mar 03 1973 15:16:40 GMT+0530 (India Standard Time)*

var d = new Date(-100000000000);*//Mon Oct 31 1966 19:43:20 GMT+0530 (India Standard Time)*

## Date Methods

Date methods allow you to get and set the year, month, day, hour, minute, second, and millisecond of date objects, using either local time or UTC (universal, or GMT) time.

## Displaying Dates

JavaScript will (by default) output dates in full text string format:

Wed Mar 25 2015 05:30:00 GMT+0530 (India Standard Time)

The toUTCString() method converts a date to a UTC string (a date display standard).

var d = new Date();*//Current time*

console.log(d.toUTCString()) *//Mon, 31 Oct 1966 14:13:20 GMT*

The toDateString() method converts a date to a more readable format:

var d = new Date();*//Current time*

console.log(d.toDateString()) *//Wed Feb 17 2021*

The toISOString() method converts a date to a string, using the ISO standard format:

var d = new Date();*//Current time*

console.log(d.toISOString()) *//2021-02-16T20:07:55.420Z*

## JavaScript Date Input

|  |  |
| --- | --- |
| ISO Date | "2015-03-25" (The International Standard) |
| Short Date | "03/25/2015" |
| Long Date | "Mar 25 2015" or "25 Mar 2015" |

## JavaScript ISO Dates

ISO 8601 is the international standard for the representation of dates and times.

The ISO 8601 syntax (YYYY-MM-DD) is also the preferred JavaScript date format:

var d = new Date("2015-03-25");

console.log(d) *//Wed Mar 25 2015 05:30:00 GMT+0530 (India Standard Time)*

## ISO Dates (Year and Month)

ISO dates can be written without specifying the day (YYYY-MM):

var d = new Date("2015-03");

console.log(d) *//Sun Mar 01 2015 05:30:00 GMT+0530 (India Standard Time)*

ISO dates can be written without month and day (YYYY):

var d = new Date("2015");

console.log(d) *//Thu Jan 01 2015 05:30:00 GMT+0530 (India Standard Time)*

## ISO Dates (Date-Time)

ISO dates can be written with added hours, minutes, and seconds (YYYY-MM-DDTHH:MM:SSZ):

var d = new Date("2015-03-25T12:00:00Z");;

console.log(d) *//Wed Mar 25 2015 17:30:00 GMT+0530 (India Standard Time)*

Date and time is separated with a capital T.

UTC time is defined with a capital letter Z.

If you want to modify the time relative to UTC, remove the Z and add +HH:MM or -HH:MM instead:

var d = new Date("2015-03-25T12:00:00-06:30");

console.log(d) *//Thu Mar 26 2015 00:00:00 GMT+0530 (India Standard Time)*

UTC (Universal Time Coordinated) is the same as GMT (Greenwich Mean Time).

## Time Zones

When setting a date, without specifying the time zone, JavaScript will use the browser's time zone.

In other words: If a date/time is created in GMT (Greenwich Mean Time), the date/time will be converted to CDT (Central US Daylight Time) if a user browses from central US.

## JavaScript Short Dates.

var d = new Date("03/25/2015");*//"MM/DD/YYYY"*

console.log(d) *//Wed Mar 25 2015 00:00:00 GMT+0530 (India Standard Time)*

## JavaScript Long Dates.

Long dates are most often written with a "MMM DD YYYY" syntax like this:

var d = new Date("Mar 25 2015");*//"MMM DD YYYY"*

var d = new Date("25 Mar 2015") *//Month and day can be in any order:*

console.log(d) *//Wed Mar 25 2015 00:00:00 GMT+0530 (India Standard Time)*

And, month can be written in full (January), or abbreviated (Jan):

var d = new Date("January 25 2015");

var d1 = new Date("Jan 25 2015");

var d2 = new Date("JANUARY, 25, 2015");*//Commas are ignored. Names are case insensitive:*

## Date Input - Parsing Dates

If you have a valid date string, you can use the Date.parse() method to convert it to milliseconds.

Date.parse() returns the number of milliseconds between the date and **January 1, 1970**:

var msec = Date.parse("March 21, 2012");

var d = new Date(msec);

console.log(msec, d) *//1332268200000 Wed Mar 21 2012 00:00:00 GMT+0530 (India Standard Time)*

You can then use the number of milliseconds to **convert it to a date** object:

JavaScript Get Date Methods

These methods can be used for getting information from a date object:

|  |  |
| --- | --- |
| **Method** | **Description** |
| getFullYear() | Get the **year** as a four digit number (yyyy) |
| getMonth() | Get the **month** as a number (0-11) |
| getDate() | Get the **day** as a number (1-31) |
| getHours() | Get the **hour** (0-23) |
| getMinutes() | Get the **minute** (0-59) |
| getSeconds() | Get the **second** (0-59) |
| getMilliseconds() | Get the **millisecond** (0-999) |
| getTime() | Get the time (milliseconds since January 1, 1970) |
| getDay() | Get the weekday as a number (0-6) |
| Date.now() | Get the time. ECMAScript 5. |

The getTime() method returns the number of milliseconds since January 1, 1970:

var date = new Date();

date.getTime(); *//1613571889395*

The getFullYear() method returns the year of a date as a four digit number:

let date = new Date();

let get = date.getFullYear(); *//2021*

The getMonth() method returns the month of a date as a number (0-11):

let date = new Date();

let get = date.getMonth(); *//1*

In JavaScript, the first month (January) is month number 0, so December returns month number 11.

The getDate() method returns the day of a date as a number (1-31)

let date = new Date();

let get = date.getDate(); *//17*

The getHours() method returns the hours of a date as a number (0-23):

let date = new Date();

let get = date.getHours(); *//20*

The getMinutes() method returns the minutes of a date as a number (0-59):

let date = new Date();

let get = date.getMinutes(); *//51*

The getSeconds() method returns the seconds of a date as a number (0-59):

let date = new Date();

let get = date.getSeconds(); *//51*

The getMilliseconds() method returns the milliseconds of a date as a number (0-999):

let date = new Date();

let get = date.getMilliseconds(); *//380*

The getDay() method returns the weekday of a date as a number (0-6):

let date = new Date();

let get = date.getDay(); *//3*

In JavaScript, the first day of the week (0) means "Sunday", even if some countries in the world consider the first day of the week to be "Monday"

## UTC Date Methods

UTC date methods are used for working with UTC dates (Universal Time Zone dates):

|  |  |
| --- | --- |
| **Method** | **Description** |
| getUTCDate() | Same as getDate(), but returns the UTC date |
| getUTCDay() | Same as getDay(), but returns the UTC day |
| getUTCFullYear() | Same as getFullYear(), but returns the UTC year |
| getUTCHours() | Same as getHours(), but returns the UTC hour |
| getUTCMilliseconds() | Same as getMilliseconds(), but returns the UTC milliseconds |
| getUTCMinutes() | Same as getMinutes(), but returns the UTC minutes |
| getUTCMonth() | Same as getMonth(), but returns the UTC month |
| getUTCSeconds() | Same as getSeconds(), but returns the UTC seconds |

## Set Date Methods

Set Date methods are used for setting a part of a date:

|  |  |
| --- | --- |
| **Method** | **Description** |
| setDate() | Set the day as a number (1-31) |
| setFullYear() | Set the year (optionally month and day) |
| setHours() | Set the hour (0-23) |
| setMilliseconds() | Set the milliseconds (0-999) |
| setMinutes() | Set the minutes (0-59) |
| setMonth() | Set the month (0-11) |
| setSeconds() | Set the seconds (0-59) |
| setTime() | Set the time (milliseconds since January 1, 1970) |

The setFullYear() method sets the year of a date object.

let date = new Date();

let get = date.setFullYear(2020); *//Mon Feb 17 2020 22:19:16 GMT+0530 (India Standard Time)*

The setFullYear() method can **optionally** set month and day:

let date = new Date();

let get = date.setFullYear(2020, 11, 3); *//Thu Dec 03 2020 22:21:48 GMT+0530 (India Standard Time)*

The setMonth() method sets the month of a date object (0-11):

let date = new Date();

let get = date.setMonth(5); *//Thu Jun 17 2021 22:23:22 GMT+0530 (India Standard Time)*

The setDate() method sets the day of a date object (1-31):

let date = new Date();

let get = date.setDate(15); *//Thu Feb 15 2021 22:23:22 GMT+0530 (India Standard Time)*

The setHours() method sets the hours of a date object (0-23):

let date = new Date();

let get = date.setHours(22); *//Wed Feb 17 2021 22:33:24 GMT+0530 (India Standard Time)*

The setMinutes() method sets the minutes of a date object (0-59):

let date = new Date();

let get = date.setMinutes(30); *//Wed Feb 17 2021 22:30:00 GMT+0530 (India Standard Time)*

The setSeconds() method sets the seconds of a date object (0-59):

let date = new Date();

let get = date.setSeconds(30); *//Wed Feb 17 2021 22:36:30 GMT+0530 (India Standard Time)*

## Compare Dates

The following example compares today's date with January 14, 2100:

var today, someday, text;

today = new Date();

someday = new Date();

someday.setFullYear(2021, 0, 14);

if (someday > today) {

  text = "Today is before January 14, 2100.";

} else {

  text = "Today is after January 14, 2100.";

}

JavaScript Math Object

The JavaScript Math object allows you to perform mathematical tasks on numbers.

Math.PI; *//3.141592653589793*

Math.round(x) returns the value of x rounded to its nearest integer:

Math.round(4.7);    *// returns 5*

Math.round(4.4);    *// returns 4*

Math.pow(x, y) returns the value of x to the power of y:

Math.pow(8, 2);      *// returns 64*

Math.sqrt(x) returns the square root of x:

Math.sqrt(64);      *// returns 8*

Math.abs(x) returns the absolute (positive) value of x:

Math.abs(-4.7);     *// returns 4.7*

Math.ceil(x) returns the value of x rounded **up** to its nearest integer:

Math.ceil(4.4);     *// returns 5*

Math.floor(x) returns the value of x rounded **down** to its nearest integer:

Math.floor(4.7);    *// returns 4*

Math.min() and Math.max() can be used to find the lowest or highest value in a list of arguments:

Math.min(0, 150, 30, 20, -8, -200);  *// returns -200*

Math.max(0, 150, 30, 20, -8, -200);  *// returns 150*

Math.random() returns a random number between 0 (inclusive), and 1 (exclusive):

Math.random();     *// returns a random number*

JavaScript provides 8 mathematical constants that can be accessed with the Math object:

Math.random();     *// returns a random number*

Math.E        *// returns Euler's number(2.718281828459045)*

Math.PI       *// returns PI(13 3.141592653589793)*

Math.SQRT2    *// returns the square root of 2(14 1.4142135623730951)*

Math.SQRT1\_2  *// returns the square root of 1/2(15 0.7071067811865476)*

Math.LN2      *// returns the natural logarithm of 2(16 0.6931471805599453)*

Math.LN10     *// returns the natural logarithm of 10(17 2.302585092994046)*

Math.LOG2E    *// returns base 2 logarithm of E(18 1.4426950408889634)*

Math.LOG10E   *// returns base 10 logarithm of E(19 0.4342944819032518)*

JavaScript Booleans

You can use the Boolean() function to find out if an expression (or a variable) is true:

Boolean(10 > 9)        *// returns true*

## Everything With a "Value" is True

100

3.14

-15

"Hello"

"false"

7 + 1 + 3.14

## Everything Without a "Value" is False

The Boolean value of **0** (zero) is **false**:

Boolean(0);       *// returns false*

The Boolean value of **-0** (minus zero) is **false**:

Boolean(-0);       *// returns false*

The Boolean value of **""**(empty string) is **false**:

Boolean("");       *// returns false*

The Boolean value of **undefined** is **false**:

Boolean(undefined);       *// returns false*

The Boolean value of **null** is **false**:

Boolean(null);       *// returns false*

The Boolean value of **NaN** is **false**:

var x = 10 / "H";

Boolean(x);       *// returns false*

JavaScript For Loop

Loops can execute a block of code a number of times.

## Different Kinds of Loops

JavaScript supports different kinds of loops:

* for - loops through a block of code a number of times
* for/in - loops through the properties of an object
* for/of - loops through the values of an iterable object
* while - loops through a block of code while a specified condition is true
* do/while - also loops through a block of code while a specified condition is true

Normally you will use statement 1 to initialize the variable used in the loop (i = 0).

This is not always the case, JavaScript doesn't care. Statement 1 is optional.

You can initiate many values in statement 1 (separated by comma):

var cars = ["BMW", "Volvo", "Saab", "Ford"];

for (var i = 0, len = cars.length, text = ""; i < len; i++) {

  text += cars[i] + "<br>";

}

And you can omit statement 1 (like when your values are set before the loop starts):

var cars = ["BMW", "Volvo", "Saab", "Ford"];

var i = 2;

var len = cars.length;

var text = "";

for (; i < len; i++) {

  text += cars[i] + "<br>";

}

Statement 3 can also be omitted (like when you increment your values inside the loop):

var i = 0;

var len = cars.length;

for (; i < len; ) {

  text += cars[i] + "<br>";

  i++;

}

JavaScript For In

The JavaScript for/in statement loops through the properties of an Object:

var person = {fname:"John", lname:"Doe", age:25};

for (let x in person) {

    console.log(x)

}

The JavaScript for/in statement can also loop over the properties of an Array:

var numbers  = [45, 4, 9, 16, 25];

for (let x in numbers ) {

    console.log(x)

}

## Array.forEach()

The forEach() method calls a function (a callback function) once for each array element.

var numbers = [45, 4, 9, 16, 25];

numbers.forEach((value, index, array) => {

    console.log(value, index, array)

  });

## The For/Of Loop

The JavaScript for/of statement loops through the values of an iterable object(such as Arrays, Strings, Maps, NodeLists, and more).

let cars = ["BMW", "Volvo", "Mini"];

for (let x of cars) {

  console.log(x)

}

JavaScript Break and Continue

The break statement "jumps out" of a loop.

The continue statement "jumps over" one iteration in the loop.

for (let i = 0; i < 10; i++) {

    if (i === 3) { break; }

    console.log(i)

}

The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

for (let i = 0; i < 10; i++) {

    if (i === 3) { continue; }

    console.log(i)

  }

JavaScript Type Conversion

## JavaScript Data Types

In JavaScript there are 5 different data types that can contain values:

* string
* number
* boolean
* object
* function

There are 6 types of objects:

* Object
* Date
* Array
* String
* Number
* Boolean

And 2 data types that cannot contain values:

* null
* undefined

You can use the typeof operator to find the data type of a JavaScript variable.

typeof "John"                 *// Returns "string"*

typeof 3.14                   *// Returns "number"*

typeof NaN                    *// Returns "number"*

typeof false                  *// Returns "boolean"*

typeof [1,2,3,4]              *// Returns "object"*

typeof {name:'John', age:34}  *// Returns "object"*

typeof new Date()             *// Returns "object"*

typeof function () {}         *// Returns "function"*

typeof myCar                  *// Returns "undefined" \**

typeof null                   *// Returns "object"*

The typeofoperator is not a variable. It is an operator.

typeof operator always **returns a string** (containing the type of the operand).

## The constructor Property

The constructor property returns the constructor function for all JavaScript variables.

"John".constructor                *// Returns function String()  {[native code]}*

    (3.14).constructor                *// Returns function Number()  {[native code]}*

false.constructor                 *// Returns function Boolean() {[native code]}*

[1, 2, 3, 4].constructor             *// Returns function Array()   {[native code]}*

{ name: 'John', age: 34 }.constructor  *// Returns function Object()  {[native code]}*

new Date().constructor            *// Returns function Date()    {[native code]}*

function () { }.constructor        *// Returns function Function(){[native code]}*

You can check the constructor property to find out if an object is an Array (contains the word "Array"):

function isArray(myArray) {

    return myArray.constructor.toString().indexOf("Array") > -1;

}

you can check if the object is an **Array function**:

function isArray(myArray) {

    return myArray.constructor === Array;

}

## Strings to Numbers

Number("3.14")    *// returns 3.14*

Number(" ")       *// returns 0*

Number("")        *// returns 0*

Number("99 88")   *// returns NaN*

## The Unary + Operator

The **unary + operator** can be used to convert a variable to a number:

var y = "5";      *// y is a string*

var x = + y;      *// x is a number*

If the variable cannot be converted, it will still become a number, but with the value NaN (Not a Number):

var y = "John";   *// y is a string*

var x = + y;      *// x is a number (NaN)*

## Converting Booleans to Numbers

The global method Number() can also convert booleans to numbers.

Number(false)     *// returns 0*

Number(true)      *// returns 1*

## Automatic Type Conversion

When JavaScript tries to operate on a "wrong" data type, it will try to convert the value to a "right" type.

5 + null    *// returns 5         because null is converted to 0*

"5" + null  *// returns "5null"   because null is converted to "null"*

"5" + 2     *// returns "52"      because 2 is converted to "2"*

"5" - 2     *// returns 3         because "5" is converted to 5*

"5" \* "2"   *// returns 10        because "5" and "2" are converted to 5 and 2*

JavaScript Regular Expressions

A regular expression is a sequence of characters that forms a search pattern.

var patt = /w3schools/i;

console.log("w3schools".toUpperCase().match(patt)); *//["W3SCHOOLS", index: 0, input: "W3SCHOOLS", groups: undefined]0: "W3SCHOOLS"groups: undefinedindex: 0input: "W3SCHOOLS"length: 1\_\_proto\_\_: Array(0)*

**/w3schools/i**  is a regular expression.

**w3schools**  is a pattern (to be used in a search).

**i**  is a modifier (modifies the search to be case-insensitive).

In JavaScript, regular expressions are often used with the two **string methods**: search() and replace().

The search() method uses an expression to search for a match, and returns the position of the match.

The replace() method returns a modified string where the pattern is replaced.

var str = "Visit W3Schools!";

var n = str.search("W3Schools"); *//6*

## Using String search() With a Regular Expression

Use a regular expression to do a case-insensitive search for "w3schools" in a string:

var str = "Visit W3Schools";

var n = str.search(/w3schools/i); *//6*

The replace() method replaces a specified value with another value in a string:

var str = "Visit Microsoft!";

var res = str.replace("Microsoft", "W3Schools"); *//Visit W3Schools!*

Use a case insensitive regular expression to replace Microsoft with W3Schools in a string:

var str = "Visit Microsoft!";

var res = str.replace(/microsoft/i, "W3Schools"); *//Visit W3Schools!*

|  |  |  |
| --- | --- | --- |
| i | Perform case-insensitive matching |  |
| g | Perform a global match (find all matches rather than stopping after the first match) |  |
| m | Perform multiline matching |  |

## Regular Expression Patterns

**Brackets** are used to find a range of characters:

|  |  |  |
| --- | --- | --- |
| [abc] | Find any of the characters between the brackets | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_abc) |
| [0-9] | Find any of the digits between the brackets | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_0-9) |
| (x|y) | Find any of the alternatives separated with | |  |

[abc]

function myFunction() {

    var str = "Is this all there is?";

    var patt1 = /[ht]/g;

    var result = str.match(patt1);

    console.log(result);

}

myFunction()  *//["t", "h", "t", "h"]*

[0-9]

function myFunction() {

    var str = "123456789";

    var patt1 = /[1-4]/g;

    var result = str.match(patt1);

    console.log(result);

}

myFunction()  *//["1", "2", "3", "4"]*

(x|y)

function myFunction() {

    var str = "re, green, red, green, gren, gr, blue, yellow";

  var patt1 = /(red|green)/g;

    var result = str.match(patt1);

    console.log(result);

}

myFunction()  *//["green", "red", "green"]*

**Metacharacters** are characters with a special meaning:

|  |  |  |
| --- | --- | --- |
| **Metacharacter** | **Description** | **Try it** |
| \d | Find a digit |  |
| \s | Find a whitespace character | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_s) |
| \b | Find a match at the beginning of a word like this: \bWORD, or at the end of a word like this: WORD\b |  |
| \uxxxx | Find the Unicode character specified by the hexadecimal number xxxx |  |

\d

function myFunction() {

    var str = "Give 100%!";

    var patt1 = /\d/g;

    var result = str.match(patt1);

    console.log(result);

}

myFunction()  *//["1", "0", "0"]*

\s

function myFunction() {

    var str = "Is this all there is?";

    var patt1 = /\s/g;

    var result = str.match(patt1);

    console.log(result);

}

myFunction()  *//[" ", " ", " ", " "]*

\b

function myFunction() {

    var str = "HELLO, LOOK AT YOU!";

    var patt1 = /\bYO/;

    var result = str.search(patt1);

    console.log(result);

}

myFunction()  *//15*

function myFunction() {

    var str = "HELLO, LOOK AT YOU!";

    var patt1 = /LO\b/;

    var result = str.search(patt1);

    console.log(result);

}

myFunction()  *//3*

\uxxxx

function myFunction() {

    var str = "Visit W3Schools. Hello World!";

    var patt1 = /\u0057/g;

    var result = str.match(patt1);

    console.log(result);

}

myFunction()  *//3*

**Quantifiers** define quantities:

|  |  |  |
| --- | --- | --- |
| **Quantifier** | **Description** | **Try it** |
| n+ | Matches any string that contains at least one *n* | [»](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_n1) |
| n\* | Matches any string that contains zero or more occurrences of *n* | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_n2) |
| n? | Matches any string that contains zero or one occurrences of *n* |  |

n+

function myFunction() {

    var str = "Hellooo World! Hello W3Schools!";

    var patt1 = /o+/g;

    var result = str.match(patt1);

    console.log(result);

}

myFunction()  *//["ooo", "o", "o", "oo"]*

n\*

function myFunction() {

    var str = "Hellooo World! Hello W3Schools!";

    var patt1 = /lo\*/g;

    var result = str.match(patt1);

    console.log(result);

}

myFunction()  *//["l", "looo", "l", "l", "lo", "l"]*

n?

function myFunction() {

    var str = "1, 100 or 1000?";

    var patt1 = /10?/g;

    var result = str.match(patt1);

    console.log(result);

}

myFunction()  *//["1", "10", "10"]*

## Using test()

The test() method is a RegExp expression method.

It searches a string for a pattern, and returns true or false, depending on the result.

function myFunction() {

    var patt = /e/;

    var result = patt.test("The best things in life are free!");

    console.log(result);

}

myFunction()  *//true*

## Using exec()

The exec() method is a RegExp expression method.

It searches a string for a specified pattern, and returns the found text as an object.

If no match is found, it returns an empty *(null)* object.

function myFunction() {

    var result = /be/.exec("The best things in life are free!");

    console.log(result);

}

myFunction()  *//["be", index: 4, input: "The best things in life are free!", groups: undefined]*

JavaScript Errors - Throw and Try to Catch

The try statement lets you test a block of code for errors.

The catch statement lets you handle the error.

The throw statement lets you create custom errors.

The finally statement lets you execute code, after try and catch, regardless of the result.

JavaScript Scope

Scope determines the accessibility (visibility) of variables.

In JavaScript there are two types of scope:

* Local scope
* Global scope

JavaScript has function scope: Each function creates a new scope.

Variables defined inside a function are not accessible (visible) from outside the function.

Variables declared within a JavaScript function, become **LOCAL** to the function.

Local variables have **Function scope**: They can only be accessed from within the function.

Since local variables are only recognized inside their functions, variables with the same name can be used in different functions.

Local variables are created when a function starts, and deleted when the function is completed.

myFunction();

function myFunction() {

  var carName = "Volvo";

  console.log(carName);

}

console.log(carName); *//ReferenceError: carName is not defined*

## Global JavaScript Variables

A variable declared outside a function, becomes **GLOBAL**.

A global variable has **global scope**: All scripts and functions on a web page can access it.

var carName = "Volvo1";

console.log(carName); *//Volvo1*

myFunction();

myFunction2();

function myFunction() {

    var carName = "Volvo2";

  console.log(carName); *//Volvo2*

}

function myFunction2() {

    var carName = "Volvo3";

    console.log(carName); *//Volvo3*

  }

## Automatically Global

If you assign a value to a variable that has not been declared, it will automatically become a **GLOBAL** variable.

myFunction();

*// code here can use carName*

function myFunction() {

  carName = "Volvo";

}

## Strict Mode

All modern browsers support running JavaScript in "Strict Mode".

In "Strict Mode", undeclared variables are not automatically global.

## Global Variables in HTML

With JavaScript, the global scope is the complete JavaScript environment.

In HTML, the global scope is the window object. All global variables belong to the window object.

## The Lifetime of JavaScript Variables

The lifetime of a JavaScript variable starts when it is declared.

Local variables are deleted when the function is completed.

In a web browser, global variables are deleted when you close the browser window (or tab).

JavaScript Hoisting

Hoisting is JavaScript's default behavior of moving declarations to the top of the current scope (to the top of the current script or the current function).

a variable can be used before it has been declared.

x = 5; *// Assign 5 to x*

console.log(x);                    *// Display x in the element*

var x; *// Declare x*

## The let and const Keywords

Variables defined with let and const are hoisted to the top of the block, but not initialized.

The block of code is aware of the variable, but it cannot be used until it has been declared.

Using a let variable before it is declared will result in a ReferenceError.

The variable is in a "temporal dead zone" from the start of the block until it is declared:

x = 5; *// ReferenceError: Cannot access 'x' before initialization*

console.log(x);

let x;

Using a const variable before it is declared, is a syntax errror, so the code will simply not run.

x = 5; *// Assign 5 to x*

console.log(x);

const x; *//SyntaxError: Missing initializer in const declarationa*

## JavaScript Initializations are Not Hoisted

JavaScript only hoists declarations, not initializations.

JavaScript in strict mode does not allow variables to be used if they are not declared.  
Study **"use strict"** in the next chapter.

JavaScript Use Strict

"use strict"; Defines that JavaScript code should be executed in "strict mode".

## The "use strict" Directive

The "use strict" directive was new in ECMAScript version 5.

The purpose of "use strict" is to indicate that the code should be executed in "strict mode".

All modern browsers support "use strict" except Internet Explorer 9 and lower:

With strict mode, you can not, for example, use undeclared variables.

You can use strict mode in all your programs. It helps you to write cleaner code, like preventing you from using undeclared variables.

**Strict mode is declared by adding "use strict"; to the beginning of a script or a function.**

"use strict";

x = 3.14;       *// ReferenceError: x is not defined :- because x is not declared*

"use strict";

myFunction();

function myFunction() {

  y = 3.14;   *// ReferenceError because y is not declared*

}

Declared inside a function, it has local scope (only the code inside the function is in strict mode):

x = 3.14;       *// This will not cause an error.*

myFunction();

function myFunction() {

  "use strict";

  y = 3.14;   *// This will cause an error*

}

## The "use strict"; Syntax

The syntax, for declaring strict mode, was designed to be compatible with older versions of JavaScript.

## Why Strict Mode?

Strict mode makes it easier to write "**secure**" JavaScript.

Strict mode changes previously accepted "bad syntax" into real errors.

As an example, in normal JavaScript, mistyping a variable name creates a new global variable. In strict mode, this will throw an error, making it impossible to accidentally create a global variable.

In strict mode, any assignment to a non-writable property, a getter-only property, a non-existing property, a non-existing variable, or a non-existing object, will throw an error.

## Not Allowed in Strict Mode

Using a variable, without declaring it, is not allowed:

Objects are variables too.

"use strict";

*// Using a variable, without declaring it, is not allowed:*

x = 3.14;                *// This will cause an error*

*// Using an object, without declaring it, is not allowed:*

x = {p1:10, p2:20};      *// This will cause an error*

*// Deleting a variable (or object) is not allowed.*

var x = 3.14;

delete x;  *// This will cause an error*

*// Deleting a function is not allowed.*

function x(p1, p2) {};

delete x;                *// This will cause an error*

*// Duplicating a parameter name is not allowed:*

function x(p1, p1) {};   *// This will cause an error*

*// Octal numeric literals are not allowed:*

var x = 010;             *// This will cause an error*

*// Octal escape characters are not allowed:*

var x = "\010";            *// This will cause an error*

*// Writing to a read-only property is not allowed:*

var obj = {};

Object.defineProperty(obj, "x", {value:0, writable:false});

obj.x = 3.14;            *// This will cause an error*

*// Writing to a get-only property is not allowed:*

var obj = {get x() {return 0} };

obj.x = 3.14;            *// This will cause an error*

*// Deleting an undeletable property is not allowed:*

delete Object.prototype; *// This will cause an error*

*// The word eval cannot be used as a variable:*

var eval = 3.14;         *// This will cause an error*

*// The word arguments cannot be used as a variable:*

var arguments = 3.14;    *// This will cause an error*

*// The with statement is not allowed:*

with (Math){x = cos(2)}; *// This will cause an error*

*// For security reasons, eval() is not allowed to create variables in the scope from which it was called:*

eval ("var x = 2");

alert (x);             *// This will cause an error*

The this keyword refers to the object that called the function.

## Future Proof!

Keywords reserved for future JavaScript versions can NOT be used as variable names in strict mode.

These are:

* implements
* interface
* let
* package
* private
* protected
* public
* static
* yield

The JavaScript **this** Keyword

The JavaScript this keyword refers to the object it belongs to.

It has different values depending on where it is used:

* In a method, this refers to the **owner object**.
* Alone, this refers to the **global object**.
* In a function, this refers to the **global object**.
* In a function, in strict mode, this is undefined.
* In an event, this refers to the **element** that received the event.
* Methods like call(), and apply() can refer this to **any object**.

## this in a Method

In an object method, this refers to the "**owner**" of the method.

var person = {

    firstName: "John",

    lastName: "Doe",

    id: 5566,

    fullName: function () {

        return this;

    }

};

*// Display data from the object:*

console.log(person.fullName()); *//{firstName: "John", lastName: "Doe", id: 5566, fullName: ƒ}*

The **person** object is the **owner** of the **fullName** method.

## this Alone

When used alone, the **owner** is the Global object, so this refers to the Global object.

 In **strict mode**, when used alone, this also refers to the Global object [object Widow]:

var x = this;

console.log(x); *//[object Window]*

## this in a Function (Default)

In a JavaScript function, the owner of the function is the **default** binding for this.

So, in a function, this refers to the Global object [object Window].

console.log(myFunction()); *// [Window object]*

function myFunction() {

  return this;

}

## this in a Function (Strict)

JavaScript **strict mode** does not allow default binding.

So, when used in a function, in strict mode, this is undefined.

"use strict"

console.log(myFunction()); *// undefined*

function myFunction() {

  return this;

}

## this in Event Handlers

In HTML event handlers, this refers to the HTML element that received the event:

<button *id*="btn" *onclick*="**this**.style.display='none'">Click to Remove Me!</button>

## Explicit Function Binding

The call() and apply() methods are predefined JavaScript methods.

They can both be used to call an object method with another object as argument.

var person1 = {

    fullName: function() {

      return this.firstName + " " + this.lastName;

    }

  }

  var person2 = {

    firstName:"John",

    lastName: "Doe",

  }

  person1.fullName.call(person2);  *// Will return "John Doe"*

JavaScript Arrow Function

Arrow functions were introduced in ES6.

hello = () => {

    return  this;  *//[window object]*

  }

It gets shorter! If the function has only one statement, and the statement returns a value, you can remove the brackets and the return keyword:

### Arrow Functions Return Value by Default:

hello = () => this;  *//[window object]*

### Arrow Function With Parameters:

hello = (value) => this + value;

console.log(hello('Hello')) *//[object Window]Hello*

if you have only one parameter, you can skip the parentheses as well

hello = value => this + value;

console.log(hello('Hello')) *//[object Window]Hello*

## What About this?

The handling of this is also different in arrow functions compared to regular functions.

with arrow functions there are no binding of this.

In regular functions the this keyword represented the object that called the function, which could be the window, the document, a button or whatever.

With arrow functions the this keyword always represents the object that defined the arrow function.

With a regular function this represents the object that calls the function:

*// Regular Function:*

hello = function() {

    console.log(this);

  }

*// The window object calls the function:*

  window.addEventListener("load", hello); *//[object Window]*

*// A button object calls the function:*

  document.getElementById("btn").addEventListener("click", hello);*//<button id="btn">click me</button>*

With an arrow function this represents the owner of the function:

*// Arrow Function:*

hello = () => console.log(this);

*// The window object calls the function:*

window.addEventListener("load", hello); *//[object Window]*

*// A button object calls the function:*

document.getElementById("btn").addEventListener("click", hello);*//[object Window]*

JavaScript Classes

ECMAScript 2015, also known as ES6, introduced JavaScript Classes.

A JavaScript class is **not** an object.

It is a **template** for JavaScript objects.

When you have a class, you can use the class to create objects:

class Car {

    constructor(name, year) {

      this.name = name;

      this.year = year;

    }

  }

  let myCar = new Car("Ford", 2014);

  console.log(`${myCar.name} ${myCar.year}`);

The constructor method is called automatically when a new object is created.

## The Constructor Method

The constructor method is a special method:

* It has to have the exact name "constructor"
* It is executed automatically when a new object is created
* It is used to initialize object properties

If you do not define a constructor method, JavaScript will add an empty constructor method.

JavaScript Best Practices

Avoid global variables, avoid new, avoid ==, avoid eval()

Minimize the use of global variables.

Global variables and functions can be overwritten by other scripts.

## Always Declare Local Variables

All variables used in a function should be declared as **local** variables.

Local variables **must** be declared with the var keyword or the let keyword, otherwise they will become global variables.

function myfunction() {

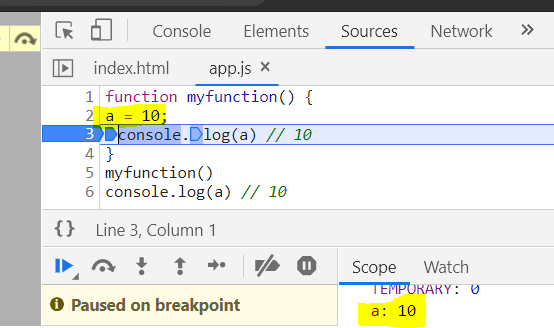
a = 10;

console.log(a) *// 10*

}

myfunction()

console.log(a) *// 10*



Strict mode does not allow undeclared variables.

## Declarations on Top

It is a good coding practice to put all declarations at the top of each script or function.

* Give cleaner code
* Provide a single place to look for local variables
* Make it easier to avoid unwanted (implied) global variables
* Reduce the possibility of unwanted re-declarations

## Initialize Variables

It is a good coding practice to initialize variables when you declare them.

This will:

* Give cleaner code
* Provide a single place to initialize variables
* Avoid undefined values

## Never Declare Number, String, or Boolean Objects

Always treat numbers, strings, or booleans as primitive values. Not as objects.

Declaring these types as objects, slows down execution speed, and produces nasty side effects:

## Don't Use new Object()

* Use {} instead of new Object()
* Use "" instead of new String()
* Use 0 instead of new Number()
* Use false instead of new Boolean()
* Use [] instead of new Array()
* Use /()/ instead of new RegExp()
* Use function (){} instead of new Function()

var x1 = {};           *// new object*

var x2 = "";           *// new primitive string*

var x3 = 0;            *// new primitive number*

var x4 = false;        *// new primitive boolean*

var x5 = [];           *// new array object*

var x6 = /()/;         *// new regexp object*

var x7 = function(){}; *// new function object*

## Beware of Automatic Type Conversions

Beware that numbers can accidentally be converted to strings or NaN (Not a Number).

JavaScript is loosely typed. A variable can contain different data types, and a variable can change its data type:

var x = "Hello";     *// typeof x is a string*

x = 5;               *// changes typeof x to a number*

var x = 5 + 7;       *// x.valueOf() is 12,  typeof x is a number*

var x = 5 + "7";     *// x.valueOf() is 57,  typeof x is a string*

var x = "5" + 7;     *// x.valueOf() is 57,  typeof x is a string*

var x = 5 - 7;       *// x.valueOf() is -2,  typeof x is a number*

var x = 5 - "7";     *// x.valueOf() is -2,  typeof x is a number*

var x = "5" - 7;     *// x.valueOf() is -2,  typeof x is a number*

var x = 5 - "x";     *// x.valueOf() is NaN, typeof x is a number*

Subtracting a string from a string, does not generate an error but returns NaN (Not a Number):

"Hello" - "Dolly"    *// returns NaN*

## Use === Comparison

The == comparison operator always converts (to matching types) before comparison.

The === operator forces comparison of values and type:

0 == false      *// true*

0 == "";        *// true*

1 == "1";       *// true*

1 == true;      *// true*

0 === false     *// false*

0 === "";       *// false*

1 === "1";      *// false*

1 === true;     *// false*

## Use Parameter Defaults

If a function is called with a missing argument, the value of the missing argument is set to undefined.

It is a good habit to assign default values to arguments.

[ECMAScript 2015](https://www.w3schools.com/js/js_es6.asp) allows default parameters in the function definition:

function (a=1, b=1) { */\*function code\*/* }

## End Your Switches with Defaults

switch (new Date().getDay()) {

    case 0:

      day = "Sunday";

      break;

    case 1:

      day = "Monday";

      break;

    default:

      day = "Unknown";

  }

## Avoid Using eval()

The eval() function is used to run text as code. In almost all cases, it should not be necessary to use it.

Because it allows arbitrary code to be run, it also represents a security problem.

JavaScript Common Mistakes

JavaScript programs may generate unexpected results if a programmer accidentally uses an assignment operator (=), instead of a comparison operator (==) in an if statement.

var x = 0;

if (x == 10)  *// false*

if (x = 10)  *//true (10 means true)*

if (x = 0)    *//false (0 means flase)*

An assignment always returns the value of the assignment.

## Expecting Loose Comparison

In regular comparison, data type does not matter. This if statement returns true:

var x = 10;

var y = "10";

console.log(x == y) *// true (data type does not matter)*

console.log(x === y) *// false (data type does matter)*

switch statements use strict comparison:

var x = 10;

switch(x) {

  case "10": alert("Hello"); *//will not display an alert*

}

switch(x) {

  case 10: alert("Hello"); *//will display an alert*

}

## Confusing Addition & Concatenation

**Addition** is about adding **numbers**.

**Concatenation** is about adding **strings**.

## Misunderstanding Floats

All numbers in JavaScript are stored as 64-bits **Floating point numbers** (Floats).

## Breaking a JavaScript String

JavaScript will allow you to break a statement into two lines:

var x =

"Hello World!";

*//breaking a statement in the middle of a string will not work:*

var x = "Hello

World!";

*// You must use a "backslash" if you must break a statement in a string:*

var x = "Hello \

World!";

## Misplacing Semicolon

Because of a misplaced semicolon, this code block will execute regardless of the value of x:

var x = 10

if (x == 19);

{

 console.log(x)

}

what will happen if you break the return statement in two lines like this:

myFunction(55); *// The function will return undefined!*

function myFunction(a) {

  var

  power = 10;

  return

  a \* power;

}

## Accessing Arrays with Named Indexes

Arrays with named indexes are called associative arrays (or hashes).

JavaScript does **not** support arrays with named indexes.

In JavaScript, **arrays** use **numbered indexes**:

In JavaScript, **objects** use **named indexes**.

JavaScript Performance

## Reduce Activity in Loops

Statements or assignments that can be placed outside the loop will make the loop run faster.

var i;

for (i = 0; i < arr.length; i++) { }  *//Bad practice*

The bad code accesses the length property of an array each time the loop is iterated.

var i;

var l = arr.length; *//  Good practice*

for (i = 0; i < l; i++) { }

The better code accesses the length property outside the loop and makes the loop run faster.

## Reduce DOM Access

Accessing the HTML DOM is very slow, compared to other JavaScript statements.

If you expect to access a DOM element several times, access it once, and use it as a local variable:

var obj;

obj = document.getElementById("demo");

obj.innerHTML = "Hello";

## Reduce DOM Size

Keep the number of elements in the HTML DOM small.

This will always improve page loading, and speed up rendering (page display), especially on smaller devices.

## Avoid Unnecessary Variables

Don't create new variables if you don't plan to save values.

var fullName = firstName + " " + lastName; *//Bad if no further use or save values of fullName variable*

document.getElementById("demo").innerHTML = fullName;

*// ------------------------------*

document.getElementById("demo").innerHTML = firstName + " " + lastName; *//Better*

## Delay JavaScript Loading

Putting your scripts at the bottom of the page body lets the browser load the page first.

While a script is downloading, the browser will not start any other downloads. In addition all parsing and rendering activity might be blocked.

An alternative is to use defer="true" in the script tag. The defer attribute specifies that the script should be executed after the page has finished parsing, but it only works for external scripts.

If possible, you can add your script to the page by code, after the page has loaded:

<script>

        window.onload = function () {

            var element = document.createElement("script");

            element.src = "myScript.js";

            document.body.appendChild(element);

        };

</script>

## Avoid Using with

Avoid using the with keyword. It has a negative effect on speed. It also clutters up JavaScript scopes.

The with keyword is **not allowed** in strict mode.

JavaScript Versions

JavaScript was invented by **Brendan Eich** in 1995, and became an ECMA standard in 1997.

ECMAScript is the official name of the language.

## ECMAScript Editions

|  |  |  |
| --- | --- | --- |
| **Ver** | **Official Name** | **Description** |
| ES1 | ECMAScript 1 (1997) | First edition |
| ES2 | ECMAScript 2 (1998) | Editorial changes |
| ES3 | ECMAScript 3 (1999) | Added regular expressions Added try/catch |
| ES4 | ECMAScript 4 | Never released |
| ES5 | ECMAScript 5 (2009)  [Read More](https://www.w3schools.com/js/js_es5.asp) | **Added "strict mode" Added JSON support Added String.trim() Added Array.isArray() Added Array iteration methods** |
| ES6 | ECMAScript 2015  [Read More](https://www.w3schools.com/js/js_es6.asp) | **Added let and const Added default parameter values Added Array.find() Added Array.findIndex()** |
|  | ECMAScript 2016  [Read More](https://www.w3schools.com/js/js_2016.asp) | **Added exponential operator (\*\*)** Added Array.prototype.includes |
|  | ECMAScript 2017  [Read More](https://www.w3schools.com/js/js_2017.asp) | Added string padding Added Object.entries  Object.entries({ a: 'sonu', b: 'sonub', c: 'sonuc' })  *// 0: (2) ["a", "sonu"]*  *// 1: (2) ["b", "sonub"]*  *// 2: (2) ["c", "sonuc"]*  Added Object.values  Object.values({a:'sonu',b:'sonub',c:'sonuc'})  *//["sonu", "sonub", "sonuc"]*  Added async functions Added shared memory |
|  | ECMAScript 2018  [Read More](https://www.w3schools.com/js/js_2018.asp) | Added rest / spread properties Added asynchronous iteration Added Promise.finally() Additions to RegExp |

ECMAScript 2009 - ES5

ECMAScript 2009, also known as ES5, was the first major revision to JavaScript.

## ES5 Features

* "use strict":- With strict mode you can, for example, not use undeclared variables.
* String.trim():-
* var str = "       Hello World!        ".trim();
* Array.isArray()
* Array.isArray(["Banana"]); *//true*
* Array.forEach()
* let sum = 0;
* console.log([45, 4, 9, 16, 25].forEach((value, index) => sum += value)) *//99*
* Array.map()
* let sum = 0;
* [45, 4, 9, 16, 25].map(value => value\*2) *//[90, 8, 18, 32, 50]*
* Array.filter()
* [45, 4, 9, 16, 25].filter(value => value>18) *//[45, 25] Creates new array*
* Array.reduce()
* [45, 4, 9, 16, 25].reduce((total,value) => total+value) *//99 reduce to single number*
* Array.reduceRight()
* [45, 4, 9, 16, 25].reduceRight((total,value) => total+value) *//99 reduce to single number*
* Array.every()
* [45, 4, 9, 16, 25].every(value => value > 18) *//false (checks if all values are over 18)*
* Array.some()
* [45, 4, 9, 16, 25].some(value => value > 18) *//false (checks if some values are over 18)*
* Array.indexOf()
* ["Banana", "Orange"].indexOf("Orange"); *//1*
* Array.lastIndexOf()
* ["Banana", "Orange"].lastIndexOf("Orange"); *//1*
* JSON.parse()
* JSON.stringify()
* Date.now()
* Property Getters and Setters
* New Object Property Methods

## Property Getters and Setters

ES5 lets you define object methods with a syntax that looks like getting or setting a property.

var person = {

    firstName: "John",

    lastName : "Doe",

    language : "NO",

    get lang() {

      return this.language;

    },

    set lang(value) {

      this.language = value.toUpperCase();

    }

  };

*// Set an object property using a setter:*

  person.lang = "en";

*// Display data from the object using a getter:*

  let a = person.lang;

console.log(a)

## New Object Property Methods

Object.defineProperty() is a new Object method in ES5.

It lets you define an object property and/or change a property's value and/or metadata.

*// Create an Object:*

var person = {

    firstName: "John",

    lastName : "Doe",

    language : "NO",

  };

*// Change a Property:*

  Object.defineProperty(person, "language", {

    value: "EN",

    writable : true,

    enumerable : true,

    configurable : true

  });

*// Enumerate Properties*

  var txt = "";

  for (var x in person) {

    txt += person[x] + "<br>";

  }

console.log(txt);

Next example is the same code, except it hides the language property from enumeration:

*// Create an Object:*

var person = {

    firstName: "John",

    lastName : "Doe",

    language : "NO",

  };

*// Change a Property:*

  Object.defineProperty(person, "language", {

    value: "EN",

    writable : true,

    enumerable : false,

    configurable : true

  });

*// Enumerate Properties*

  var txt = "";

  for (var x in person) {

    txt += person[x] + "<br>";

  }

console.log(txt);

This example creates a setter and a getter to secure upper case updates of language:

*/// Create an Object:*

var person = {

    firstName: "John",

    lastName : "Doe",

    language : "NO"

  };

*// Change a Property:*

  Object.defineProperty(person, "language", {

    get : function() { return language },

    set : function(value) { language = value.toUpperCase()}

  });

*// Change Language*

  person.language = "en";

*// Display Language*

 console.log(person.language);

ES5 added a lot of new Object Methods to JavaScript:

### ES5 New Object Methods

*// Adding or changing an object property*

Object.defineProperty(object, property, descriptor)

*// Adding or changing many object properties*

Object.defineProperties(object, descriptors)

*// Accessing Properties*

Object.getOwnPropertyDescriptor(object, property)

*// Returns all properties as an array*

Object.getOwnPropertyNames(object)

*// Returns enumerable properties as an array*

Object.keys(object)

*// Accessing the prototype*

Object.getPrototypeOf(object)

*// Prevents adding properties to an object*

Object.preventExtensions(object)

*// Returns true if properties can be added to an object*

Object.isExtensible(object)

*// Prevents changes of object properties (not values)*

Object.seal(object)

*// Returns true if object is sealed*

Object.isSealed(object)

*// Prevents any changes to an object*

Object.freeze(object)

*// Returns true if object is frozen*

Object.isFrozen(object)

## Trailing Commas

ES5 allows trailing commas in object and array definitions:

person = {

    firstName: "John",

    lastName: " Doe",

    age: 46,

  }

## Strings Over Multiple Lines

ES5 allows string literals over multiple lines if escaped with a backslash:

var a = "Hello \

Dolly!";

ES5 allows reserved words as property names:

var obj = {name: "John", new: "yes"}

console.log(obj)

ECMAScript 2015 - ES6

ECMAScript 6 was the second major revision to JavaScript.

## New Features in ES6

* [The let keyword](https://www.w3schools.com/js/js_es6.asp#mark_let)
* [The const keyword](https://www.w3schools.com/js/js_es6.asp#mark_const)
* [JavaScript Arrow Functions](https://www.w3schools.com/js/js_es6.asp#mark_arrow)
* [JavaScript For/of](https://www.w3schools.com/js/js_es6.asp#mark_forof)
* [JavaScript Classes](https://www.w3schools.com/js/js_es6.asp#mark_class)
* [JavaScript Promises](https://www.w3schools.com/js/js_es6.asp#mark_promise)
* [JavaScript Symbol](https://www.w3schools.com/js/js_es6.asp#mark_symbol)
* [Default Parameters](https://www.w3schools.com/js/js_es6.asp#mark_param)
* [Function Rest Parameter](https://www.w3schools.com/js/js_es6.asp#mark_rest)
* [Array.find()](https://www.w3schools.com/js/js_es6.asp#mark_array_find)
* [Array.findIndex()](https://www.w3schools.com/js/js_es6.asp#mark_array_findIndex)
* [New Number Properties](https://www.w3schools.com/js/js_es6.asp#mark_number_properties)
* [New Number Methods](https://www.w3schools.com/js/js_es6.asp#mark_number_methods)
* [New Global Methods](https://www.w3schools.com/js/js_es6.asp#mark_global_methods)
* JavaScript Modules

## JavaScript let

The let keyword allows you to declare a variable with block scope.

## JavaScript const

Constants are similar to let variables, except that the value cannot be changed.

## Arrow Functions

Arrow functions allows a short syntax for writing function expressions.

var x = function(x, y) {

    return x \* y;

 }

*// ES6*

 const x = (x, y) => x \* y;

**Arrow functions do not have their own this**. They are not well suited for defining **object methods**.

Arrow functions are not hoisted. They must be defined **before** they are used.

Using const is safer than using var, because a function expression is always a constant value.

## The For/Of Loop

The JavaScript for/of statement loops through the values of an iterable objects.

for/of lets you loop over data structures that are iterable such as Arrays, Strings, Maps, NodeLists, and more.

var cars = ["BMW", "Volvo", "Mini"];

var x;

for (x of cars) {

  console.log(x);

}

## JavaScript Classes

A JavaScript class is **not** an object.

It is a **template** for JavaScript objects.

class Car {

    constructor(name, year) {

      this.name = name;

      this.year = year;

    }

  }

## JavaScript Promises

A Promise is a JavaScript object that links "Producing Code" and "Consuming Code".

let myPromise = new Promise(function(myResolve, myReject) {

*// "Producing Code" (May take some time)*

      myResolve(); *// when successful*

      myReject();  *// when error*

    });

*// "Consuming Code" (Must wait for a fulfilled Promise).*

    myPromise.then(

      function(value) { */\* code if successful \*/* },

      function(error) { */\* code if some error \*/* }

    );

**Example**

let myPromise = new Promise(function (myResolve, myReject) {

    setTimeout(function () { console.log(this); myResolve("I love You !!"); }, 3000);

    setTimeout(() => {

        console.log(this);

        myReject("I don't love you");

    }, 2000);

});

myPromise.then(function (value) {

    console.log(value);

}, (value) => {

    console.log(value)

});

## The Symbol Type

A JavaScript Symbol is a primitive datatype just like Number, String, or Boolean.

It represents a unique "hidden" identifier that no other code can accidentally access.

For instance, if different coders want to add a person.id property to a person object belonging to a third-party code, they could mix each others values.

Using Symbol() to create a unique identifiers, solves this problem:

const person = {

    firstName: "John",

    lastName: "Doe",

    age: 50,

    eyeColor: "blue"

  };

  let id = Symbol('id');

  person.id = 140353;

  console.log(id) *//Symbol(id)*

Symbols are always unique.

If you create two symbols with the same description they will have different values.

## Default Parameter Values

ES6 allows function parameters to have default values.

function myFunction(x, y = 10) {

*// y is 10 if not passed or undefined*

    return x + y;

}

myFunction(5); *// will return 15*

## Function Rest Parameter

The rest parameter (...) allows a function to treat an indefinite number of arguments as an array:

function getSum(...args) {

    let sum = 0;

    for (let arg of args) sum += arg;

    return sum;

  }

  let x = getSum(4, 9, 16, 25, 29, 100, 66, 77);

## Array.find()

The find() method returns the value of the first array element that passes a test function.

## Array.findIndex()

The findIndex() method returns the index of the first array element that passes a test function.

## New Number Properties

ES6 added the following properties to the Number object:

* EPSILON
* MIN\_SAFE\_INTEGER
* MAX\_SAFE\_INTEGER
* console.log(Number.EPSILON)*//2.220446049250313e-16*
* console.log(Number.MIN\_SAFE\_INTEGER)*//-9007199254740991*
* console.log(Number.MAX\_SAFE\_INTEGER)*//9007199254740991*

## New Number Methods

ES6 added 2 new methods to the Number object:

* Number.isInteger()
* Number.isSafeInteger()

The Number.isInteger() method returns true if the argument is an integer.

Number.isInteger(10);        *// returns true*

Number.isInteger(10.5);      *// returns false*

The Number.isSafeInteger() method returns true if the argument is a safe integer.

Number.isSafeInteger(10);    *// returns true*

Number.isSafeInteger(12345678901234567890);  *// returns false*

Safe integers are all integers from **-(253 - 1) to +(253 - 1).**  
This is safe: 9007199254740991. This is not safe: 9007199254740992.

## New Global Methods

ES6 added 2 new global number methods:

* isFinite()
* isNaN()

## The isFinite() Method

The global isFinite() method returns false if the argument is Infinity or NaN.

Otherwise it returns true:

isFinite(10/0);       *// returns false*

isFinite(10/1);       *// returns true*

## The isNaN() Method

The global isNaN() method returns true if the argument is NaN. Otherwise it returns false:

isNaN("Hello");       *// returns true*

ECMAScript 2016

## New Features in ECMAScript 2016

* JavaScript Exponentiation (\*\*)
* JavaScript Exponentiation assignment (\*\*=)
* JavaScript Array.prototype.includes

The **exponentiation** operator (\*\*) raises the first operand to the power of the second operand.

let x = 5;

let z = x \*\* 2;          *// result is 25*

x \*\* y produces the same result as Math.pow(x, y):

The **exponentiation assignment** operator (\*\*=) raises the value of a variable to the power of the right operand.

let x = 5;

x \*\*= 2; *// result 25*

## JavaScript Array.prototype.includes

ECMAScript 2016 introduced Array.prototype.includes to arrays. This allows us to check if an element is present in an array:

const fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.includes("Mango"); *// is true*

ECMAScript 2017

## New Features in ECMAScript 2017

This chapter introduces the new features in ECMAScript 2017:

* [JavaScript String padding](https://www.w3schools.com/js/js_2017.asp#mark_padding)
* [JavaScript Object.entries](https://www.w3schools.com/js/js_2017.asp#mark_obj_entries)
* [JavaScript Object.values](https://www.w3schools.com/js/js_2017.asp#mark_obj_values)
* [JavaScript async functions](https://www.w3schools.com/js/js_2017.asp#mark_async)
* JavaScript shared memory

## JavaScript String Padding

ECMAScript 2017 added two String methods: padStart and padEnd to support padding at the beginning and at the end of a string.

let str = "5";

str = str.padStart(4,0);

*// result is 0005*

let str = "5";

str = str.padEnd(4,0);

*// result is 5000*

## JavaScript Object Entries

ECMAScript 2017 adds a new Object.entries method to objects:

const person = {

    firstName : "John",

    lastName  : "Doe",

    age     : 50,

    eyeColor  : "blue"

  };

console.log(Object.entries(person));

*//0: (2) ["firstName", "John"]*

*// 1: (2) ["lastName", "Doe"]*

*// 2: (2) ["age", 50]*

*// 3: (2) ["eyeColor", "blue"]*

## JavaScript Object Values

Object.values are similar to Object.entries, but returns a single dimension array of the object values:

const person = {

    firstName : "John",

    lastName  : "Doe",

    age     : 50,

    eyeColor  : "blue"

  };

console.log(Object.values(person));*//["John", "Doe", 50, "blue"]*

## JavaScript Async Functions

*async* function myDisplay() {

    let myPromise = new Promise(function (myResolve, myReject) {

        setTimeout(function () { myResolve("I love You !!"); }, 3000);

    });

    let result = await myPromise;

    console.log(result)

}

myDisplay();

ECMAScript 2018

## New Features in ECMAScript 2018

* [Asynchronous Iteration](https://www.w3schools.com/js/js_2018.asp#mark_async_iteration)
* [Promise Finally](https://www.w3schools.com/js/js_2018.asp#mark_promise_finally)
* [Object Rest Properties](https://www.w3schools.com/js/js_2018.asp#mark_obj_rest)
* [New RegExp Features](https://www.w3schools.com/js/js_2018.asp#mark_regxp)

## JavaScript Asynchronous Iteration

ECMAScript 2018 added asynchronous iterators and iterables.

With asynchronous iterables, we can use the await keyword in for/of loops.

for await () {}

## JavaScript Promise.finally

ECMAScript 2018 finalizes the full implermentation of the Promise object with Promise.finally:

let myPromise = new Promise();

myPromise.then();

myPromise.catch();

myPromise.finally();

*async* function myDisplay() {

    let myPromise = new Promise(function (myResolve, myReject) {

        setTimeout(function () { myResolve("I love You !!"); }, 3000);

        setTimeout(function () { myReject("I don't love You !!"); }, 2000);

    });

    myPromise.then((value) => console.log(value), (reason) => console.log(reason));

    myPromise.catch((value) => console.log(value));

    myPromise.finally((value) => console.log(value));

}

myDisplay();

## JavaScript Object Rest Properties

ECMAScript 2018 added rest properties.

This allows us to destruct an object and collect the leftovers onto a new object:

let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };

console.log(x); *// 1*

console.log(y); *// 2*

console.log(z); *// { a: 3, b: 4 }*

## New JavaScript RegExp Features

ECMAScript 2018 added 4 new RegExp features:

* Unicode Property Escapes (\p{...})
* Lookbehind Assertions (?<= ) and (?<! )
* Named Capture Groups
* s (dotAll) Flag

## JavaScript Form Validation

HTML form validation can be done by JavaScript.

If a form field (fname) is empty, this function alerts a message, and returns false, to prevent the form from being submitted:

function validateForm() {

    var x = document.forms["myForm"]["fname"].value;

    if (x == "") {

      alert("Name must be filled out");

      return false;

    }

  }

 <form *name*="myForm" *action*="/action\_page.php" *onsubmit*="return validateForm()" *method*="post">

        Name: <input *type*="text" *name*="fname">

        <input *type*="submit" *value*="Submit">

    </form>

## Automatic HTML Form Validation

HTML form validation can be performed automatically by the browser:

If a form field (fname) is empty, the **required** attribute prevents this form from being submitted:

<form *action*="/action\_page.php" *method*="post">

        <input *type*="text" *name*="fname" *required*>

        <input *type*="submit" *value*="Submit">

    </form>

## Data Validation

Data validation is the process of ensuring that user input is clean, correct, and useful.

Typical validation tasks are:

* has the user filled in all required fields?
* has the user entered a valid date?
* has the user entered text in a numeric field?

**Server side validation** is performed by a web server, after input has been sent to the server.

**Client side validation** is performed by a web browser, before input is sent to a web server.

## HTML Constraint Validation

HTML5 introduced a new HTML validation concept called **constraint validation**.

HTML constraint validation is based on:

* Constraint validation **HTML** **Input Attributes**
* Constraint validation **CSS Pseudo Selectors**
* Constraint validation **DOM Properties and Methods**

JavaScript Validation API

## Constraint Validation DOM Methods

|  |  |
| --- | --- |
| **Property** | **Description** |
| checkValidity() | Returns true if an input element contains valid data. |
| setCustomValidity() | Sets the validationMessage property of an input element. |

<input *id*="id1" *type*="number" *min*="100" *max*="300" *required*>

    <button *onclick*="myFunction()">OK</button>

    <p *id*="demo"></p>

function myFunction() {

    var inpObj = document.getElementById("id1");

    if (!inpObj.checkValidity()) {

      document.getElementById("demo").innerHTML = inpObj.validationMessage;

    }

  }

## Constraint Validation DOM Properties

|  |  |
| --- | --- |
| **Property** | **Description** |
| validity | Contains boolean properties related to the validity of an input element. |
| validationMessage | Contains the message a browser will display when the validity is false. |
| willValidate | Indicates if an input element will be validated. |

## Validity Properties

The **validity property** of an input element contains a number of properties related to the validity of data:

## Validity Properties

The **validity property** of an input element contains a number of properties related to the validity of data:

|  |  |
| --- | --- |
| **Property** | **Description** |
| customError | Set to true, if a custom validity message is set. |
| patternMismatch | Set to true, if an element's value does not match its pattern attribute. |
| rangeOverflow | Set to true, if an element's value is greater than its max attribute. |
| rangeUnderflow | Set to true, if an element's value is less than its min attribute. |
| stepMismatch | Set to true, if an element's value is invalid per its step attribute. |
| tooLong | Set to true, if an element's value exceeds its maxLength attribute. |
| typeMismatch | Set to true, if an element's value is invalid per its type attribute. |
| valueMissing | Set to true, if an element (with a required attribute) has no value. |
| valid | Set to true, if an element's value is valid. |

function myFunction() {

    var inpObj = document.getElementById("id1");

    console.log(inpObj.validity)

*//badInput: false*

*// customError: false*

*// patternMismatch: false*

*// rangeOverflow: false*

*// rangeUnderflow: true*

*// stepMismatch: false*

*// tooLong: false*

*// tooShort: false*

*// typeMismatch: false*

*// valid: false*

*// valueMissing: false*

    console.log(inpObj.validationMessage)

    console.log(inpObj.willValidate)

    if (!inpObj.checkValidity()) {

        document.getElementById("demo").innerHTML = inpObj.validationMessage;

    }

}

**Example**

<input *id*="id1" *type*="number" *max*="100">

    <button *onclick*="myFunction()">OK</button>

    <p *id*="demo"></p>

function myFunction() {

    var txt = "";

    if (document.getElementById("id1").validity.rangeOverflow) {

      txt = "Value too large";

    }

    document.getElementById("demo").innerHTML = txt;

  }

function myFunction() {

    var txt = "";

    if (document.getElementById("id1").validity.rangeOverflow) {

      txt = "Value too large"+'\t '+document.getElementById("id1").validationMessage;

    }

    document.getElementById("demo").innerHTML = txt;

  }

JavaScript Objects

In JavaScript, almost "everything" is an object.

* Booleans can be objects (if defined with the new keyword)
* Numbers can be objects (if defined with the new keyword)
* Strings can be objects (if defined with the new keyword)
* Dates are always objects
* Maths are always objects
* Regular expressions are always objects
* Arrays are always objects
* Functions are always objects
* Objects are always objects

All JavaScript values, except primitives, are objects.

## JavaScript Primitives

A **primitive value** is a value that has no properties or methods.

A **primitive data type** is data that has a primitive value.

JavaScript defines 5 types of primitive data types:

* string
* number
* boolean
* null
* undefined

Primitive values are immutable (they are hardcoded and therefore cannot be changed).

if x = 3.14, you can change the value of x. But you cannot change the value of 3.14.

|  |  |  |
| --- | --- | --- |
| **Value** | **Type** | **Comment** |
| "Hello" | string | "Hello" is always "Hello" |
| 3.14 | number | 3.14 is always 3.14 |
| true | boolean | true is always true |
| false | boolean | false is always false |
| null | null (object) | null is always null |
| undefined | undefined | undefined is always undefined |

## Objects are Variables

JavaScript variables can contain single values:

var person = "John Doe";

Objects are variables too. But objects can contain many values.

var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};

A JavaScript object is a collection of **named values**

The named values, in JavaScript objects, are called **properties**.

## Object Methods

Methods are **actions** that can be performed on objects.

Object properties can be both primitive values, other objects, and functions.

An **object method** is an object property containing a **function definition**.

JavaScript objects are containers for **named values**, called properties and methods.

## Creating a JavaScript Object

There are different ways to create new objects:

* Define and create a single object, using an object literal.
* Define and create a single object, with the keyword new.
* Define an object constructor, and then create objects of the constructed type.

In ECMAScript 5, an object can also be created with the function Object.create().

## Using an Object Literal

This is the easiest way to create a JavaScript Object.

Using an object literal, you both define and create an object in one statement.

An object literal is a list of name:value pairs (like age:50) inside curly braces {}.

var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};

## Using the JavaScript Keyword new

var person = new Object();

person.firstName = "John";

person.lastName = "Doe";

There is no need to use new Object().  
For simplicity, readability and execution speed, use the first one (the object literal method).

## JavaScript Objects are Mutable

Objects are mutable: They are addressed by reference, not by value.

If person is an object, the following statement will not create a copy of person:

var x = person;  // This will not create a copy of person.

The object x is **not a copy** of person. It **is** person. Both x and person are the same object.

Any changes to x will also change person, because x and person are the same object.

var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"}

var x = person;

x.age = 10;           *// This will change both x.age and person.age*

JavaScript Object Properties

Properties are the most important part of any JavaScript object.

Properties are the values associated with a JavaScript object.

A JavaScript object is a collection of unordered properties.

## Accessing JavaScript Properties

objectName.property         *// person.age*

objectName["property"]      *// person["age"]*

objectName[expression]      *// x = "age"; person[x]*

The expression must evaluate to a property name.

## JavaScript for...in Loop

The JavaScript for...in statement loops through the properties of an object.

for (variable in object) {

*// code to be executed*

}

## Adding New Properties

person.nationality = "English";

## Deleting Properties

The delete keyword deletes a property from an object:

var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};

delete person.age;   *// or delete person["age"];*

The delete operator is designed to be used on object properties. It has no effect on variables or functions.

## Prototype Properties

JavaScript objects inherit the properties of their prototype.

The delete keyword does not delete inherited properties, but if you delete a prototype property, it will affect all objects inherited from the prototype.

JavaScript Object Methods

[❮ Previous](https://www.w3schools.com/js/js_object_properties.asp)[Next ❯](https://www.w3schools.com/js/js_object_display.asp)

var person = {

    firstName: "John",

    lastName: "Doe",

    id: 5566,

**fullName**: function () {

        return this.firstName + " " + this.lastName;

    }

};

## The this Keyword

In a function definition, this refers to the "owner" of the function.

In the example above, this is the **person object** that "owns" the **fullName** function.

In other words, **this.firstName** means the **firstName** property of **this object**.

## JavaScript Methods

A JavaScript **method** is a property containing a **function definition**.

Methods are functions stored as object properties.

## Accessing Object Methods

*objectName.methodName()*

You will typically describe fullName() as a method of the person object, and fullName as a property.

name = person.fullName();

## Adding a Method to an Object

Adding a new method to an object is easy:

person.name = function () {

    return this.firstName + " " + this.lastName;

};

## How to Display JavaScript Objects?

Displaying a JavaScript object will output **[object Object]**.

Some common solutions to display JavaScript objects are:

* Displaying the Object Properties by name
* Displaying the Object Properties in a Loop
* Displaying the Object using Object.values()
* Displaying the Object using JSON.stringify()

## Using Object.values()

Any JavaScript object can be converted to an array using Object.values():

var person = { name: "John", age: 30, city: "New York" };

var myArray = Object.values(person);

console.log(myArray)*// ["John", 30, "New York"]*

## Using JSON.stringify()

Any JavaScript object can be stringified (converted to a string) with the JavaScript function JSON.stringify():

var person = { name: "John", age: 30, city: "New York" };

var myString = JSON.stringify(person);

console.log(myString) *//{"name":"John","age":30,"city":"New York"}*

## Stringify Dates

JSON.stringify converts dates into strings:

## Stringify Functions

JSON.stringify will not stringify functions:

var person = {name:"John", age:function () {return 30;}};

var myString = JSON.stringify(person);

console.log(myString); *//{"name":"John"}*

This can be "fixed" if you convert the functions into strings before stringifying.

var person = {name:"John", age:function () {return 30;}};

person.age = person.age.toString();

var myString = JSON.stringify(person);

console.log(myString); *//{"name":"John","age":"function () {return 30;}"}*

## Stringify Arrays

var arr = ["John", "Peter", "Sally", "Jane"];

var myString = JSON.stringify(arr);

console.log(myString); *//["John","Peter","Sally","Jane"]*

JavaScript Object Accessors

ECMAScript 5 (2009) introduced Getter and Setters.

Getters and setters allow you to define Object Accessors (Computed Properties).

## JavaScript Getter (The get Keyword)

*// Create an object:*

var person = {

    firstName: "John",

    lastName : "Doe",

    language : "en",

    get lang() {

      return this.language;

    }

  };

*// Display data from the object using a getter:*

console.log(person.lang); *//en*

## JavaScript Setter (The set Keyword)

*// Create an object:*

var person = {

    firstName: "John",

    lastName : "Doe",

    language : "",

    set lang(lang) {

      this.language = lang;

    }

  };

*// Set an object property using a setter:*

  person.lang = "en";

*// Display data from the object:*

console.log(person.language); *//en*

## JavaScript Function or Getter?

var person = {

    firstName: "John",

    lastName: "Doe",

    fullName: function () {

        return this.firstName + " " + this.lastName;

    }

};

*// Display data from the object using a method:*

console.log(person.fullName()); *//en*

var person = {

    firstName: "John",

    lastName: "Doe",

    get fullName() {

        return this.firstName + " " + this.lastName;

    }

};

*// Display data from the object using a method:*

console.log(person.fullName); *//en*

Example 1 access fullName as a function: person.fullName().

Example 2 access fullName as a property: person.fullName.

The second example provides simpler syntax.

## Data Quality

JavaScript can secure better data quality when using getters and setters.

Using the lang property, in this example, returns the value of the language property in upper case:

*// Create an object:*

var person = {

    firstName: "John",

    lastName : "Doe",

    language : "en",

    get lang() {

      return this.language.toUpperCase();

    }

  };

*// Display data from the object using a getter:*

console.log(person.lang); *//en*

Using the lang property, in this example, stores an upper case value in the language property:

var person = {

    firstName: "John",

    lastName: "Doe",

    language: "",

    set lang(lang) {

        this.language = lang.toUpperCase();

    }

};

*// Set an object property using a setter:*

person.lang = "en";

console.log(person.language); *//en*

## Why Using Getters and Setters?

* It gives simpler syntax
* It allows equal syntax for properties and methods
* It can secure better data quality
* It is useful for doing things behind-the-scenes

## Object.defineProperty()

The Object.defineProperty() method can also be used to add Getters and Setters:

*// Define object*

var obj = { counter: 0 };

console.log(obj);

*// Define setters*

Object.defineProperty(obj, "reset", {

    get: function () { this.counter = 0; }

});

Object.defineProperty(obj, "increment", {

    get: function () { this.counter++; }

});

Object.defineProperty(obj, "decrement", {

    get: function () { this.counter--; }

});

Object.defineProperty(obj, "add", {

    set: function (value) { this.counter += value; }

});

Object.defineProperty(obj, "subtract", {

    set: function (value) { this.counter -= value; }

});

*// Play with the counter:*

console.log(obj);

console.log(obj.reset);

console.log(obj.add = 5);

console.log(obj.subtract = 1);

console.log(obj.increment);

console.log(obj.decrement);

JavaScript Object Constructors

function Person(first, last, age, eye) {

    this.firstName = first;

    this.lastName = last;

    this.age = age;

    this.eyeColor = eye;

}

It is considered good practice to name constructor functions with an upper-case first letter.

## Object Types (Blueprints) (Classes)

Sometimes we need a "**blueprint**" for creating many objects of the same "type".

The way to create an "object type", is to use an **object constructor function**.

In the example above, function Person() is an object constructor function.

Objects of the same type are created by calling the constructor function with the new keyword:

function Person(first, last, age, eye) {

    this.firstName = first;

    this.lastName = last;

    this.age = age;

    this.eyeColor = eye;

}

var myFather = new Person("John", "Doe", 50, "blue");

var myMother = new Person("Sally", "Rally", 48, "green");

console.log(myFather) *//Person {firstName: "John", lastName: "Doe", age: 50, eyeColor: "blue"}*

console.log(myMother) *//Person {firstName: "Sally", lastName: "Rally", age: 48, eyeColor: "green"}*

## The ****this**** Keyword

In JavaScript, the thing called this is the object that "owns" the code.

The value of this, when used in an object, is the object itself.

In a constructor function this does not have a value. It is a substitute for the new object. The value of this will become the new object when a new object is created.

## Adding a Property to an Object

myFather.nationality = "English";

## Adding a Method to an Object

myFather.name = function () {

    return this.firstName + " " + this.lastName;

};

## Built-in JavaScript Constructors

JavaScript has built-in constructors for native objects:

var x1 = new Object();    *// A new Object object*

var x2 = new String();    *// A new String object*

var x3 = new Number();    *// A new Number object*

var x4 = new Boolean();   *// A new Boolean object*

var x5 = new Array();     *// A new Array object*

var x6 = new RegExp();    *// A new RegExp object*

var x7 = new Function();  *// A new Function object*

var x8 = new Date();      *// A new Date object*

The Math() object is not in the list. Math is a global object. The new keyword cannot be used on Math.

## Did You Know?

As you can see above, JavaScript has object versions of the primitive data types String, Number, and Boolean. But there is no reason to create complex objects. Primitive values are much faster.

Use object literals {} instead of new Object().

Use string literals "" instead of new String().

Use number literals 12345 instead of new Number().

Use boolean literals true / false instead of new Boolean().

Use array literals [] instead of new Array().

Use pattern literals /()/ instead of new RegExp().

Use function expressions () {} instead of new Function().

var x1 = {};            *// new object*

var x2 = "";            *// new primitive string*

var x3 = 0;             *// new primitive number*

var x4 = false;         *// new primitive boolean*

var x5 = [];            *// new array object*

var x6 = /()/           *// new regexp object*

var x7 = function(){};  *// new function object*

## String Objects

Normally, strings are created as primitives: var firstName = "John"

But strings can also be created as objects using the new keyword: var firstName = new String("John")

## Number Objects

Normally, numbers are created as primitives: var x = 123

But numbers can also be created as objects using the new keyword: var x = new Number(123)

## Boolean Objects

Normally, booleans are created as primitives: var x = false

But booleans can also be created as objects using the new keyword: var x = new Boolean(false)

JavaScript Object Prototypes

All JavaScript objects inherit properties and methods from a prototype.

## Prototype Inheritance

All JavaScript objects inherit properties and methods from a prototype:

* Date objects inherit from Date.prototype
* Array objects inherit from Array.prototype
* Person objects inherit from Person.prototype

The Object.prototype is on the top of the prototype inheritance chain:

Date objects, Array objects, and Person objects inherit from Object.prototype.

## Adding Properties and Methods to Objects

Sometimes you want to add new properties (or methods) to all existing objects of a given type.

Sometimes you want to add new properties (or methods) to an object constructor.

## Using the ****prototype**** Property

The JavaScript prototype property allows you to add new properties to object constructors:

function Person(first, last, age, eyecolor) {

    this.firstName = first;

    this.lastName = last;

    this.age = age;

    this.eyeColor = eyecolor;

  }

  Person.prototype.nationality = "English";

  console.log((new Person(1,2,3,4)).nationality)

The JavaScript prototype property also allows you to add new methods to objects constructors:

function Person(first, last, age, eyecolor) {

    this.firstName = first;

    this.lastName = last;

    this.age = age;

    this.eyeColor = eyecolor;

  }

  Person.prototype.name = function() {

    return this.firstName + " " + this.lastName;

  };

  console.log((new Person('Mojahid' , 'Islam', 33, 'black').name())) *// Mojahid Islam*

Only modify your **own** prototypes. Never modify the prototypes of standard JavaScript objects.

JavaScript ES5 Object Methods

ECMAScript 5 added a lot of new Object Methods to JavaScript.

### ES5 New Object Methods

*// Adding or changing an object property*

Object.defineProperty(object, property, descriptor)

*// Adding or changing many object properties*

Object.defineProperties(object, descriptors)

*// Accessing Properties*

Object.getOwnPropertyDescriptor(object, property)

*// Returns all properties as an array*

Object.getOwnPropertyNames(object)

*// Returns enumerable properties as an array*

Object.keys(object)

*// Accessing the prototype*

Object.getPrototypeOf(object)

*// Prevents adding properties to an object*

Object.preventExtensions(object)

*// Returns true if properties can be added to an object*

Object.isExtensible(object)

*// Prevents changes of object properties (not values)*

Object.seal(object)

*// Returns true if object is sealed*

Object.isSealed(object)

*// Prevents any changes to an object*

Object.freeze(object)

*// Returns true if object is frozen*

Object.isFrozen(object)

## Changing a Property Value

**Object.defineProperty(object, property, {value : value})**

var person = {

    firstName: "John",

    lastName : "Doe",

    language : "EN"

  };

*// Change a property*

  Object.defineProperty(person, "language", {value : "NO"});

  console.log(person)

## Changing Meta Data

ES5 allows the following property meta data to be changed:

writable : true      *// Property value can be changed*

enumerable : true    *// Property can be enumerated*

configurable : true  *// Property can be reconfigured*

writable : false     *// Property value can not be changed*

enumerable : false   *// Property can be not enumerated*

configurable : false *// Property can be not reconfigured*

ES5 allows getters and setters to be changed:

*// Defining a getter*

get: function() { return language }

*// Defining a setter*

set: function(value) { language = value }

This example makes language read-only:

Object.defineProperty(person, "language", {writable:false});

This example makes language not enumerable:

Object.defineProperty(person, "language", {enumerable:false});

## Listing All Properties

var person = {

    firstName: "John",

    lastName: "Doe",

    language: "EN"

};

Object.defineProperty(person, "language", { enumerable: false });

let props = Object.getOwnPropertyNames(person);  *// Returns an array of properties*

console.log(props) *//["firstName", "lastName", "language"]*

## Listing Enumerable Properties

var person = {

    firstName: "John",

    lastName: "Doe",

    language: "EN"

};

Object.defineProperty(person, "language", { enumerable: false });

let enumProps = Object.keys(person);  *// Returns an array of enumerable properties*

console.log(enumProps)

|  | **in** | **for..in** | **obj.hasOwnProperty** | **obj.propertyIsEnumerable** | **Object.keys** | **Object.getOwnPropertyNames** | **Object.getOwnPropertyDescriptors** | **Reflect.ownKeys()** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Enumerable** | true | true | true | true | true | true | true | true |
| **Nonenumerable** | true | false | true | false | false | true | true | true |
| **Symbols keys** | true | false | true | true | false | false | true | true |
| **Inherited Enumerable** | true | true | false | false | false | false | false | false |
| **Inherited Nonenumerable** | true | false | false | false | false | false | false | false |
| **Inherited Symbols keys** | true | false | false | false | false | false | false | false |

## Adding a Property

*// Create an object:*

var person = {

    firstName: "John",

    lastName: "Doe",

    language: "EN"

};

*// Add a property*

let enumProps = Object.defineProperty(person, "year", { value: "2008" });

console.log(enumProps) *//{firstName: "John", lastName: "Doe", language: "EN", year: "2008"}*

## Adding Getters and Setters

The Object.defineProperty() method can also be used to add Getters and Setters:

*//Create an object*

var person = { firstName: "John", lastName: "Doe" };

*// Define a getter*

let enumProps = Object.defineProperty(person, "fullName", {

    get: function () { return this.firstName + " " + this.lastName; }

});

console.log(enumProps.fullName) *//John Doe*

JavaScript Function Definitions

JavaScript functions are **defined** with the function keyword.

You can use a function **declaration** or a function **expression**.

## Function Declarations

Earlier in this tutorial, you learned that functions are **declared** with the following syntax:

function functionName(parameters) {

*// code to be executed*

}

Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are invoked (called upon).

function myFunction(a, b) {

    return a \* b;

}

Semicolons are used to separate executable JavaScript statements.  
Since a function **declaration** is not an executable statement, it is not common to end it with a semicolon.

## Function Expressions

A JavaScript function can also be defined using an **expression**.

A function expression can be stored in a variable:

var x = function (a, b) {return a \* b};

var z = x(4, 3);

The function above is actually an **anonymous function** (a function without a name).

Functions stored in variables do not need function names. They are always invoked (called) using the variable name.

The function above ends with a semicolon because it is a part of an executable statement.

## The Function() Constructor

Functions can also be defined with a built-in JavaScript function constructor called Function().

var myFunction = new Function("a", "b", "return a \* b");

var x = myFunction(4, 3);

Most of the time, you can avoid using the new keyword in JavaScript.

## Function Hoisting

Hoisting is JavaScript's default behavior of moving **declarations** to the top of the current scope.

Hoisting applies to variable declarations and to function declarations.

Because of this, JavaScript functions can be called before they are declared:

console.log(myFunction(5)); *//25*

function myFunction(y) {

  return y \* y;

}

Functions defined using an expression are not hoisted.

## Self-Invoking Functions

A self-invoking expression is invoked (started) automatically, without being called

Function expressions will execute automatically if the expression is followed by ().

You have to add parentheses around the function to indicate that it is a function expression:

(function () {

    console.log("Hello!!");  *// I will invoke myself*

})();

The function above is actually an **anonymous self-invoking function** (function without name).

## Functions Can Be Used as Values

JavaScript functions can be used as values:

function myFunction(a, b) {

    return a \* b;

}

var x = myFunction(4, 3);

JavaScript functions can be used in expressions:

function myFunction(a, b) {

    return a \* b;

}

var x = myFunction(4, 3) \* 2;

## Functions are Objects

The typeof operator in JavaScript returns "function" for functions.

But, JavaScript functions can best be described as objects.

JavaScript functions have both **properties** and **methods**.

The arguments.length property returns the number of arguments received when the function was invoked:

function myFunction(a, b) {

    return arguments.length;

}

var x = myFunction(4, 3);

console.log(x) *// 2*

A function defined as the property of an object, is called a method to the object.  
A function designed to create new objects, is called an object constructor.

## Arrow Functions

Arrow functions allows a short syntax for writing function expressions.

You don't need the function keyword, the return keyword, and the **curly brackets**.

*// ES5*

var x = function (x, y) {

    return x \* y;

}

*// ES6*

const x = (x, y) => x \* y;

Arrow functions do not have their own this. They are not well suited for defining **object methods**.

Arrow functions are not hoisted. They must be defined **before** they are used.

Using const is safer than using var, because a function expression is always constant value.

JavaScript Function Parameters

A JavaScript function does not perform any checking on parameter values (arguments).

## Function Parameters and Arguments

Function **parameters** are the **names** listed in the function definition.

Function **arguments** are the real **values** passed to (and received by) the function.

## Parameter Rules

JavaScript function definitions do not specify data types for parameters.

JavaScript functions do not perform type checking on the passed arguments.

JavaScript functions do not check the number of arguments received.

## Default Parameters

If a function is called with **missing arguments** (less than declared), the missing values are set to undefined.

Sometimes this is acceptable, but sometimes it is better to assign a default value to the parameter:

function (x, y = 2) {

*// function code*

}

## The Arguments Object

JavaScript functions have a built-in object called the arguments object.

The argument object contains an array of the arguments used when the function was called (invoked).

x = findMax(1, 123, 500, 115, 44, 88);

function findMax() {

  var i;

  var max = -Infinity;

  for (i = 0; i < arguments.length; i++) {

    if (arguments[i] > max) {

      max = arguments[i];

    }

  }

  return max;

}

console.log(x) *//500*

Or create a function to sum all input values:

x = sumAll(1, 123, 500, 115, 44, 88);

function sumAll() {

  var i;

  var sum = 0;

  for (i = 0; i < arguments.length; i++) {

    sum += arguments[i];

  }

  return sum;

}

## Arguments are Passed by Value

The parameters, in a function call, are the function's arguments.

JavaScript arguments are passed by **value**: The function only gets to know the values, not the argument's locations.

If a function changes an argument's value, it does not change the parameter's original value.

**Changes to arguments are not visible (reflected) outside the function.**

## Objects are Passed by Reference

If a function changes an object property, it changes the original value.

**Changes to object properties are visible (reflected) outside the function.**

JavaScript Function Invocation

The code inside a JavaScript function will execute when "something" invokes it.

## Invoking a JavaScript Function

The code inside a function is not executed when the function is **defined**.

The code inside a function is executed when the function is **invoked**.

It is common to use the term "**call a function**" instead of "**invoke a function**".

## Invoking a Function as a Function

function myFunction(a, b) {

    return a \* b;

  }

  myFunction(10, 2);

The function above does not belong to any object. But in JavaScript there is always a default global object.

In HTML the default global object is the HTML page itself, so the function above "belongs" to the HTML page.

In a browser the page object is the browser window. The function above automatically becomes a window function.

myFunction() and window.myFunction() is the same function:

function myFunction(a, b) {

    return a \* b;

}

let x = window.myFunction(10, 2);    *// Will also return 20*

console.log(x);

## The *****this***** Keyword

In JavaScript, the thing called this, is the object that "owns" the current code.

The value of this, when used in a function, is the object that "owns" the function.

## The Global Object

When a function is called without an owner object, the value of this becomes the global object.

In a web browser the global object is the browser window.

This example returns the window object as the value of this:

var x = myFunction();            *// x will be the window object*

function myFunction() {

    return this;

}

console.log(x.toString()) *//[object Window]*

## Invoking a Function as a Method

In JavaScript you can define functions as object methods.

The following example creates an object (**myObject**), with two properties (**firstName** and **lastName**), and a method (**fullName**):

var myObject = {

    firstName: "John",

    lastName: "Doe",

    fullName: function () {

        return this.firstName + " " + this.lastName;

    }

}

myObject.fullName();         *// Will return "John Doe"*

The **fullName** method is a function. The function belongs to the object. **myObject** is the owner of the function.

var myObject = {

    firstName: "John",

    lastName: "Doe",

    fullName: function () {

        return this;

    }

}

let x = myObject.fullName();          *// Will return [object Object] (the owner object)*

console.log(x)

## Invoking a Function with a Function Constructor

If a function invocation is preceded with the new keyword, it is a constructor invocation.

It looks like you create a new function, but since JavaScript functions are objects you actually create a new object:

*// This is a function constructor:*

function myFunction(arg1, arg2) {

    this.firstName = arg1;

    this.lastName = arg2;

}

*// This creates a new object*

var x = new myFunction("John", "Doe");

x.firstName;                             *// Will return "John"*

JavaScript Function Call

## Method Reuse

With the **call()** method, you can write a method that can be used on different objects.

With call(), an object can use a method belonging to another object.

var person = {

    fullName: function() {

      return this.firstName + " " + this.lastName;

    }

  }

  var person1 = {

    firstName:"John",

    lastName: "Doe"

  }

  var person2 = {

    firstName:"Mary",

    lastName: "Doe"

  }

  person.fullName.call(person1);  *// Will return "John Doe"*

## The call() Method with Arguments

var person = {

    fullName: function (city, country) {

        return this.firstName + " " + this.lastName + "," + city + "," + country;

    }

}

var person1 = {

    firstName: "John",

    lastName: "Doe"

}

person.fullName.call(person1, "Oslo", "Norway");

## Method Reuse

With the apply() method, you can write a method that can be used on different objects.

## The JavaScript apply() Method

The apply() method is similar to the call() method

var person = {

    fullName: function() {

      return this.firstName + " " + this.lastName;

    }

  }

  var person1 = {

    firstName: "Mary",

    lastName: "Doe"

  }

  person.fullName.apply(person1);  *// Will return "Mary Doe"*

## The Difference Between call() and apply()

The difference is:

The call() method takes arguments **separately**.

The apply() method takes arguments as an **array**.

The apply() method is very handy if you want to use an array instead of an argument list.

The apply() method accepts arguments in an array:

var person = {

    fullName: function (city, country) {

        return this.firstName + " " + this.lastName + "," + city + "," + country;

    }

}

var person1 = {

    firstName: "John",

    lastName: "Doe"

}

let x = person.fullName.apply(person1, ["Oslo", "Norway"]);

## Simulate a Max Method on Arrays

Math.max(1,2,3);  *// Will return 3*

Since JavaScript **arrays** do not have a max() method, you can apply the Math.max() method instead.

Math.max.apply(null, [1,2,3]);  *// Will return 3*

The first argument (null) does not matter. It is not used in this example.

Math.max.apply(Math, [1,2,3])

Math.max.apply(" ", [1,2,3])

Math.max.apply(0, [1,2,3])

## JavaScript Strict Mode

In JavaScript strict mode, if the first argument of the apply() method is not an object, it becomes the owner (object) of the invoked function. In "non-strict" mode, it becomes the global object.

JavaScript Closures

JavaScript variables can belong to the **local** or **global** scope.

Global variables can be made local (private) with **closures**.

## Global Variables

A function can access all variables defined **inside** the function, like this:

function myFunction() {

    var a = 4;

    return a \* a;

}

console.log(a); *// ReferenceError: a is not defined*

console.log(myFunction()); *//16*

But a function can also access variables defined **outside** the function, like this:

var a = 4;

function myFunction() {

    return a \* a;

}

console.log(a); *// 4*

console.log(myFunction()); *//16*

In the last example, **a** is a **global** variable.

In a web page, global variables belong to the window object.

Global variables can be used (and changed) by all scripts in the page (and in the window).

In the first example, **a** is a **local** variable.

A local variable can only be used inside the function where it is defined. It is hidden from other functions and other scripting code.

Global and local variables with the same name are different variables. Modifying one, does not modify the other.

Variables created **without** a declaration keyword (var, let, or const) are always global, even if they are created inside a function.

## Variable Lifetime

Global variables live until the page is discarded, like when you navigate to another page or close the window.

Local variables have short lives. They are created when the function is invoked, and deleted when the function is finished.

## A Counter Dilemma

Suppose you want to use a variable for counting something, and you want this counter to be available to all functions.

You could use a global variable, and a function to increase the counter:

*// Initiate counter*

var counter = 0;

*// Function to increment counter*

function add() {

  counter += 1;

}

*// Call add() 3 times*

add();

add();

add();

console.log(counter)*// The counter should now be 3*

There is a problem with the solution above: Any code on the page can change the counter, without calling add().

The counter should be local to the add() function, to prevent other code from changing it:

*// Initiate counter*

var counter = 0;

*// Function to increment counter*

function add() {

  var counter = 0;

  counter += 1;

}

*// Call add() 3 times*

add();

add();

add();

*//The counter should now be 3. But it is 0*

console.log(counter);

It did not work because we display the global counter instead of the local counter.

We can remove the global counter and access the local counter by letting the function return it:

*// Function to increment counter*

function add() {

    var counter = 0;

    counter += 1;

    return counter;

}

*// Call add() 3 times*

add();

add();

add();

console.log(add());

*//The counter should now be 3. But it is 1.*

It did not work because we reset the local counter every time we call the function.

**A JavaScript inner function can solve this.**

## JavaScript Nested Functions

All functions have access to the global scope.

In fact, in JavaScript, all functions have access to the scope "above" them.

JavaScript supports nested functions. Nested functions have access to the scope "above" them.

In this example, the inner function plus() has access to the counter variable in the parent function:

function add() {

    var counter = 0;

    function plus() { counter += 1; }

    plus();

    return counter;

}

*// Call add() 3 times*

add();

add();

add();

console.log(add());

This could have solved the counter dilemma, if we could reach the plus() function from the outside.

We also need to find a way to execute counter = 0 only once.

**We need a closure.**

## JavaScript Closures

Remember self-invoking functions? What does this function do?

var add = (function () {

    var counter = 0;

    return function () { counter += 1; return counter }

})()

add();

add();

add();

*// the counter is now 3*

console.log(add());

## Example Explained

The variable add is assigned to the return value of a self-invoking function.

The self-invoking function only runs once. It sets the counter to zero (0), and returns a function expression.

This way add becomes a function. The "wonderful" part is that it can access the counter in the parent scope.

This is called a JavaScript **closure.** It makes it possible for a function to have "**private**" variables.

The counter is protected by the scope of the anonymous function, and can only be changed using the add function.

A closure is a function having access to the parent scope, even after the parent function has closed.

JavaScript Classes

ECMAScript 2015, also known as ES6, introduced JavaScript Classes.

JavaScript Classes are templates not object for JavaScript Objects.

## JavaScript Class Syntax

Use the keyword class to create a class.

Always add a method named constructor():

When you have a class, you can use the class to create objects:

The constructor method is called automatically when a new object is created.

## The Constructor Method

The constructor method is a special method:

* It has to have the exact name "constructor"
* It is executed automatically when a new object is created
* It is used to initialize object properties

If you do not define a constructor method, JavaScript will add an empty constructor method.

JavaScript Class Inheritance

To create a class inheritance, use the extends keyword.

A class created with a class inheritance inherits all the methods from another class:

class Car {

    constructor(brand) {

        this.carname = brand;

    }

    present() {

        return 'I have a ' + this.carname;

    }

}

class Model *extends* Car {

    constructor(brand, mod) {

        super(brand);

        this.model = mod;

    }

    show() {

        return this.present() + ', it is a ' + this.model;

    }

}

let myCar = new Model("Ford", "Mustang");

console.log(myCar.show());

The super() method refers to the parent class.

By calling the super() method in the constructor method, we call the parent's constructor method and gets access to the parent's properties and methods.

Inheritance is useful for code reusability: reuse properties and methods of an existing class when you create a new class.

## Getters and Setters

Classes also allows you to use getters and setters.

It can be smart to use getters and setters for your properties, especially if you want to do something special with the value before returning them, or before you set them.

To add getters and setters in the class, use the get and set keywords.

class Car {

    constructor(brand) {

        this.carname = brand;

    }

    get cnam() {

        return this.carname;

    }

    set cnam(x) {

        this.carname = x;

    }

}

let myCar = new Car("Ford");

console.log(myCar.cnam);

You can use the underscore character to separate the getter/setter from the actual property:

class Car {

    constructor(brand) {

        this.\_carname = brand;

    }

    get carname() {

        return this.\_carname;

    }

    set carname(x) {

        this.\_carname = x;

    }

}

let myCar = new Car("Ford");

console.log(myCar.\_carname);

Use a setter to change the carname to "Volvo":

class Car {

    constructor(brand) {

        this.\_carname = brand;

    }

    get carname() {

        return this.\_carname;

    }

    set carname(x) {

        this.\_carname = x;

    }

}

let myCar = new Car("Ford");

myCar.carname = "Volvo";

console.log(myCar.carname);

## Hoisting

Unlike functions, and other JavaScript declarations, class declarations are not hoisted.

That means that you must declare a class before you can use it:

*//You cannot use the class yet.*

myCar = new Car("Ford") *//ReferenceError: Cannot access 'Car' before initialization*

*//This would raise an error.*

class Car {

    constructor(brand) {

        this.carname = brand;

    }

}

*//Now you can use the class:*

let myCar = new Car("Ford")

JavaScript Static Methods

You cannot call a static method on an object, only on an object class.

If you want to use the myCar object inside the static method, you can send it as a parameter:

class Car {

    constructor(name) {

        this.name = name;

    }

*static* hello(x) {

        return "Hello " + x.name;

    }

}

let myCar = new Car("Ford");

console.log(Car.hello(myCar));

JavaScript Callbacks

*"I will call back later!"*

A callback is a function passed as an argument to another function

This technique allows a function to call another function

A callback function can run after another function has finished

## Function Sequence

JavaScript functions are executed in the sequence they are called. Not in the sequence they are defined.

## Sequence Control

Sometimes you would like to have better control over when to execute a function.

Suppose you want to do a calculation, and then display the result.

Or, you could call a calculator function (myCalculator), and let the calculator function call the display function (myDisplayer):

function myDisplayer(some) {

    document.getElementById("demo").innerHTML = some;

  }

  function myCalculator(num1, num2) {

    let sum = num1 + num2;

    myDisplayer(sum);

  }

  myCalculator(5, 5);

The problem with the second example, is that you cannot prevent the calculator function from displaying the result.

Now it is time to bring in a callback.

## JavaScript Callbacks

A callback is a function passed as an argument to another function.

Using a callback, you could call the calculator function (myCalculator) with a callback, and let the calculator function run the callback after the calculation is finished:

function myDisplayer(some) {

    document.getElementById("demo").innerHTML = some;

  }

  function myCalculator(num1, num2, myCallback) {

    let sum = num1 + num2;

    myCallback(sum);

  }

  myCalculator(5, 5, myDisplayer);

## When to Use a Callback?

Where callbacks really shine are in asynchronous functions, where one function has to wait for another function (like waiting for a file to load).

Asynchronous JavaScript

*I will finish later!*

Functions running in parallel with other functions are called asynchronous

A good example is JavaScript setTimeout()

## Asynchronous JavaScript

## Waiting for a Timeout

When using the JavaScript function setTimeout(), you can specify a callback function to be executed on time-out:

setTimeout(myFunction, 3000);

function myFunction() {

    console.log("I love You !!");

}

## Waiting for Intervals:

When using the JavaScript function setInterval(), you can specify a callback function to be executed for each interval:

setInterval(myFunction, 1000);

function myFunction() {

    let d = new Date();

    console.log(`${d.getHours()} ${d.getMinutes()} ${d.getSeconds()}`);

}

## Waiting for Files

If you create a function to load an external resource (like a script or a file), you cannot use the content before it is fully loaded.

function myDisplayer(some) {

    document.getElementById("demo").innerHTML = some;

}

function getFile(myCallback) {

    let req = new XMLHttpRequest();

    req.open('GET', "index.html");

    req.onload = function () {

        if (req.status == 200) {

            myCallback(this.responseText);

        } else {

            myCallback("Error: " + req.status);

        }

    }

    req.send();

}

getFile(myDisplayer);

JavaScript Promises

*"I Promise a Result!"*

"Producing code" is code that can take some time

"Consuming code" is code that must wait for the result

A Promise is a JavaScript object that links producing code and consuming code

## JavaScript Promise Object

A JavaScript Promise object contains both the producing code and calls to the consuming code:

let myPromise = new Promise(function (myResolve, myReject) {

*// "Producing Code" (May take some time)*

    myResolve(); *// when successful*

    myReject();  *// when error*

});

*// "Consuming Code" (Must wait for a fulfilled Promise)*

myPromise.then(

    function (value) { */\* code if successful \*/* },

    function (error) { */\* code if some error \*/* }

);

When the executing code obtains the result, it should call one of the two callbacks:

|  |  |
| --- | --- |
| **Result** | **Call** |
| Success | myResolve(result value) |
| Error | myReject(error object) |

## Promise Object Properties

A JavaScript Promise object can be:

* Pending
* Fulfilled
* Rejected
* The Promise object supports two properties: **state** and **result**.
* While a Promise object is "pending" (working), the result is undefined.
* When a Promise object is "fulfilled", the result is a value.
* When a Promise object is "rejected", the result is an error object.

|  |  |
| --- | --- |
| **myPromise.state** | **myPromise.result** |
| "pending" | undefined |
| "fulfilled" | a result value |
| "rejected" | an error object |

You cannot access the Promise properties **state** and **result**.

You must use a Promise method to handle promises.

## Promise How To

Here is how to use a Promise:

myPromise.then(

    function (value) { */\* code if successful \*/* },

    function (error) { */\* code if some error \*/* }

);

Promise.then() takes two arguments, a callback for success and another for failure.

Both are optional, so you can add a callback for success or failure only.

function myDisplayer(some) {

    document.getElementById("demo").innerHTML = some;

  }

  let myPromise = new Promise(function(myResolve, myReject) {

    let x = 0;

*// The producing code (this may take some time)*

    if (x == 0) {

      myResolve("OK");

    } else {

      myReject("Error");

    }

  });

  myPromise.then(

    function(value) {myDisplayer(value);},

    function(error) {myDisplayer(error);}

  );

## Waiting for a Timeout

let myPromise = new Promise(function (myResolve, myReject) {

    setTimeout(function () { myResolve("I love You !!"); }, 3000);

});

myPromise.then(function (value) {

    document.getElementById("demo").innerHTML = value;

});

## Waiting for a file

let myPromise = new Promise(function (myResolve, myReject) {

    let req = new XMLHttpRequest();

    req.open('GET', "index.html");

    req.onload = function () {

        if (req.status == 200) {

            myResolve(req.response);

        } else {

            myReject("File not Found");

        }

    };

    req.send();

});

myPromise.then(

    function (value) { console.log(value); },

    function (error) { console.log(error); }

);

JavaScript Async

*"async and await make promises easier to write"*

**async** makes a function return a Promise

**await** makes a function wait for a Promise

## Async Syntax

The keyword async before a function makes the function return a promise:

*async* function myFunction() {

    return "Hello";

}

*//   Is the same as:*

*async* function myFunction() {

    return Promise.resolve("Hello");

}

### Example

*async* function myFunction() {

    return "Hello";

}

myFunction().then(

    function (value) { console.log(value); },

    function (error) { console.log(error); }

);

## Await Syntax

The keyword await before a function makes the function wait for a promise:

The await keyword can only be used inside an async function.

### Waiting for a Timeout

*async* function myDisplay() {

    let myPromise = new Promise(function (myResolve, myReject) {

        setTimeout(function () { myResolve("I love You !!"); }, 3000);

    });

    document.getElementById("demo").innerHTML = await myPromise;

}

myDisplay();

JavaScript HTML DOM

With the HTML DOM, JavaScript can access and change all the elements of an HTML document.

## The HTML DOM (Document Object Model)

When a web page is loaded, the browser creates a **D**ocument **O**bject **M**odel of the page.

The **HTML DOM** model is constructed as a tree of **Objects**: 

With the object model, JavaScript gets all the power it needs to create dynamic HTML:

* JavaScript can change all the HTML elements in the page
* JavaScript can change all the HTML attributes in the page
* JavaScript can change all the CSS styles in the page
* JavaScript can remove existing HTML elements and attributes
* JavaScript can add new HTML elements and attributes
* JavaScript can react to all existing HTML events in the page
* JavaScript can create new HTML events in the page

## What is the DOM?

The DOM is a W3C (World Wide Web Consortium) standard.

The DOM defines a standard for accessing documents:

*"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."*

The W3C DOM standard is separated into 3 different parts:

* Core DOM - standard model for all document types
* XML DOM - standard model for XML documents
* HTML DOM - standard model for HTML documents

## What is the HTML DOM?

The HTML DOM is a standard **object** model and **programming interface** for HTML. It defines:

* The HTML elements as **objects**
* The **properties** of all HTML elements
* The **methods** to access all HTML elements
* The **events** for all HTML elements

In other words:**The HTML DOM is a standard for how to get, change, add, or delete HTML elements.**

JavaScript - HTML DOM Methods

HTML DOM methods are **actions** you can perform (on HTML Elements).

HTML DOM properties are **values** (of HTML Elements) that you can set or change.

## The DOM Programming Interface

The HTML DOM can be accessed with JavaScript (and with other programming languages).

In the DOM, all HTML elements are defined as **objects**.

A **property** is a value that you can get or set (like changing the content of an HTML element).

A **method** is an action you can do (like add or deleting an HTML element).

In the example above, getElementById is a **method**, while innerHTML is a **property**.

The most common way to access an HTML element is to use the id of the element.

The easiest way to get the content of an element is by using the innerHTML property.

The innerHTML property is useful for getting or replacing the content of HTML elements.

JavaScript HTML DOM Document

The HTML DOM document object is the owner of all other objects in your web page.

The document object represents your web page.

If you want to access any element in an HTML page, you always start with accessing the document object.

## Changing HTML Elements

|  |  |
| --- | --- |
| **Property** | **Description** |
| *element*.innerHTML =  *new html content* | Change the inner HTML of an element |
| *element*.*attribute = new value* | Change the attribute value of an HTML element |
| *element*.style.*property = new style* | Change the style of an HTML element |
| **Method** | **Description** |
| *element*.setAttribute*(attribute, value)* | Change the attribute value of an HTML element |

## Adding and Deleting Elements

|  |  |
| --- | --- |
| **Method** | **Description** |
| document.createElement(*element*) | Create an HTML element |
| document.removeChild(*element*) | Remove an HTML element |
| document.appendChild(*element*) | Add an HTML element |
| document.replaceChild(*new, old*) | Replace an HTML element |
| document.write(*text*) | Write into the HTML output stream |

## Adding Events Handlers

|  |  |
| --- | --- |
| **Method** | **Description** |
| document.getElementById(*id*).onclick = function(){*code*} | Adding event handler code to an onclick event |

## Finding HTML Objects

document.getElementById('demo').onclick = () => console.log('clicked');

console.log(document.anchors); *//Returns all <a> elements that have a name attribute*

console.log(document.baseURI); *//Returns the absolute base URI of the document*

console.log(document.body); *//Returns the <body> element*

console.log(document.cookie); *//    Returns the document's cookie*

console.log(document.doctype); *//   Returns the document's doctype*

console.log(document.documentElement); *//   Returns the <html> element*

console.log(document.documentMode); *//  Returns the mode used by the browser*

console.log(document.documentURI); *//   Returns the URI of the document*

console.log(document.domain); *//    Returns the domain name of the document server*

console.log(document.embeds); *//    Returns all <embed> elements*

console.log(document.forms); *// Returns all <form> elements*

console.log(document.head); *//  Returns the <head> element*

console.log(document.images); *//    Returns all <img> elements*

console.log(document.implementation); *//    Returns the DOM implementation*

console.log(document.inputEncoding); *// Returns the document's encoding (character set)*

console.log(document.lastModified); *//  Returns the date and time the document was updated*

console.log(document.links); *// Returns all <area> and <a> elements that have a href attribute*

console.log(document.readyState); *//    Returns the (loading) status of the document*

console.log(document.referrer); *//  Returns the URI of the referrer (the linking document)*

console.log(document.scripts); *//   Returns all <script> elements*

console.log(document.strictErrorChecking); *//   Returns if error checking is enforced*

console.log(document.title); *// Returns the <title> element*

console.log(document.URL); *//   Returns the complete URL of the document*

JavaScript HTML DOM Elements

This page teaches you how to find and access HTML elements in an HTML page.

## Finding HTML Elements

Often, with JavaScript, you want to manipulate HTML elements.

To do so, you have to find the elements first. There are several ways to do this:

* Finding HTML elements by id
* Finding HTML elements by tag name
* Finding HTML elements by class name
* Finding HTML elements by CSS selectors
* Finding HTML elements by HTML object collections

## Finding HTML Element by Id

var myElement = document.getElementById("intro");

If the element is found, the method will return the element as an object (in myElement).

If the element is not found, myElement will contain null.

## Finding HTML Elements by Tag Name

var myElement = document.getElementsByTagName("p");

This example finds the element with id="main", and then finds all <p> elements inside "main":

var x = document.getElementById("main");

var y = x.getElementsByTagName("p");

## Finding HTML Elements by Class Name

If you want to find all HTML elements with the same class name, use getElementsByClassName().

This example returns a list of all elements with class="intro".

var x = document.getElementsByClassName("intro");

Finding elements by class name does not work in Internet Explorer 8 and earlier versions.

## Finding HTML Elements by CSS Selectors

If you want to find all HTML elements that match a specified CSS selector (id, class names, types, attributes, values of attributes, etc), use the querySelectorAll() method.

var x = document.querySelectorAll("p.intro");

The querySelectorAll() method does not work in Internet Explorer 8 and earlier versions.

JavaScript HTML DOM - Changing CSS

The HTML DOM allows JavaScript to change the style of HTML elements.

## Changing HTML Style

document.getElementById(*id*).style.*property*=*new style*

document.getElementById("demo").style.color = "blue";

## Using Events

The HTML DOM allows you to execute code when an event occurs.

Events are generated by the browser when "things happen" to HTML elements:

* An element is clicked on
* The page has loaded
* Input fields are changed

<h1 *id*="id1">My Heading 1</h1>

    <button *type*="button" *onclick*="document.getElementById('id1').style.color = 'red'">

        Click Me!</button>

JavaScript HTML DOM Animation

All animations should be relative to a container element.

## Style the Elements

The container element should be created with style = "position: relative".

The animation element should be created with style = "position: absolute".

## Animation Code

JavaScript animations are done by programming gradual changes in an element's style.

The changes are called by a timer. When the timer interval is small, the animation looks continuous.

<style>

    #container {

        width: 400px;

        height: 400px;

        position: relative;

        background: yellow;

    }

    #animate {

        width: 50px;

        height: 50px;

        position: absolute;

        background-color: red;

    }

</style>

<body>

    <p><button *onclick*="myMove()">Click Me</button></p>

    <div *id*="container">

        <div *id*="animate"></div>

    </div>

    <script>

        var id = null;

        function myMove() {

            var elem = document.getElementById("animate");

            var pos = 0;

            clearInterval(id);

            id = setInterval(frame, 5);

            function frame() {

                if (pos == 350) {

                    clearInterval(id);

                } else {

                    pos++;

                    elem.style.top = pos + "px";

                    elem.style.left = pos + "px";

                }

            }

        }

    </script>

JavaScript HTML DOM Events

HTML DOM allows JavaScript to react to HTML events:

## Reacting to Events

A JavaScript can be executed when an event occurs, like when a user clicks on an HTML element.

To execute code when a user clicks on an element, add JavaScript code to an HTML event attribute:

Examples of HTML events:

* When a user clicks the mouse
* When a web page has loaded
* When an image has been loaded
* When the mouse moves over an element
* When an input field is changed
* When an HTML form is submitted
* When a user strokes a key

## The onload and onunload Events

The onload and onunload events are triggered when the user enters or leaves the page.

<body onload="checkCookies()">

## The onchange Event

The onchange event is often used in combination with validation of input fields.

<input *type*="text" *id*="fname" *onchange*="myFunction()">

## The addEventListener() method

Add an event listener that fires when a user clicks a button:

document.getElementById("btn").addEventListener("click", ()=>console.log('Button clicked'));

The addEventListener() method attaches an event handler to the specified element.

The addEventListener() method attaches an event handler to an element without overwriting existing event handlers.

You can add many event handlers of the same type to one element, i.e two "click" events.

You can add event listeners to any DOM object not only HTML elements. i.e the window object.

The addEventListener() method makes it easier to control how the event reacts to bubbling.

When using the addEventListener() method, the JavaScript is separated from the HTML markup, for better readability and allows you to add event listeners even when you do not control the HTML markup.

You can easily remove an event listener by using the removeEventListener() method.

element.addEventListener(event, function, useCapture);

The third parameter is a boolean value specifying whether to use event bubbling or event capturing. This parameter is optional.

## Add Many Event Handlers to the Same Element

The addEventListener() method allows you to add many events to the same element, without overwriting existing events:

element.addEventListener("click", myFunction);  
element.addEventListener("click", mySecondFunction);

You can add events of different types to the same element:

    document.getElementById("btn").addEventListener("click", ()=>console.log('Button clicked'));

    document.getElementById("btn").addEventListener("mouseover", ()=>console.log('Button mouseover'));

    document.getElementById("btn").addEventListener("mouseout", ()=>console.log('Button mouseout'));

## Add an Event Handler to the window Object

The addEventListener() method allows you to add event listeners on any HTML DOM object such as HTML elements, the HTML document, the window object, or other objects that support events, like the xmlHttpRequest object.

window.addEventListener("resize", function (event) {

    console.log(event)

    document.getElementById("btn").innerHTML = event.target.screenX;

});

## Event Bubbling or Event Capturing?

There are two ways of event propagation in the HTML DOM, bubbling and capturing.

Event propagation is a way of defining the element order when an event occurs. If you have a <p> element inside a <div> element, and the user clicks on the <p> element, which element's "click" event should be handled first?

In bubbling the inner most element's event is handled first and then the outer: the <p> element's click event is handled first, then the <div> element's click event.

In capturing the outer most element's event is handled first and then the inner: the <div> element's click event will be handled first, then the <p> element's click event.

With the addEventListener() method you can specify the propagation type by using the "useCapture" parameter:

addEventListener(event, function, useCapture);

The default value is false, which will use the bubbling propagation, when the value is set to true, the event uses the capturing propagation.

## The removeEventListener() method

The removeEventListener() method removes event handlers that have been attached with the addEventListener() method:

element.removeEventListener("mousemove", myFunction);

JavaScript Window - The Browser Object Model

The Browser Object Model (BOM) allows JavaScript to "talk to" the browser.

There are no official standards for the **B**rowser **O**bject **M**odel (BOM).

## The Window Object

The window object is supported by all browsers. It represents the browser's window.

All global JavaScript objects, functions, and variables automatically become members of the window object.

Global variables are properties of the window object.

Global functions are methods of the window object.

Even the document object (of the HTML DOM) is a property of the window object:

window.document.getElementById("header");

is the same as:

document.getElementById("header");

## Window Size

Two properties can be used to determine the size of the browser window.

* window.innerHeight - the inner height of the browser window (in pixels)
* window.innerWidth - the inner width of the browser window (in pixels)

For Internet Explorer 8, 7, 6, 5:

* document.documentElement.clientHeight
* document.documentElement.clientWidth
* or
* document.body.clientHeight
* document.body.clientWidth

A practical JavaScript solution (covering all browsers):

var w = window.innerWidth

|| document.documentElement.clientWidth

|| document.body.clientWidth;

var h = window.innerHeight

|| document.documentElement.clientHeight

|| document.body.clientHeight;

console.log(w,h)

## Other Window Methods

Some other methods:

## Other Window Methods

Some other methods:

* window.open() - open a new window
* window.close() - close the current window
* window.moveTo() - move the current window
* window.resizeTo() - resize the current window
* JavaScript Window Screen

The window.screen object contains information about the user's screen.

The window.screen object can be written without the window prefix.

Properties:

* screen.width
* screen.height
* screen.availWidth
* screen.availHeight
* screen.colorDepth
* screen.pixelDepth

The screen.availWidth property returns the width of the visitor's screen, in pixels, minus interface features like the Windows Taskbar.

The screen.colorDepth property returns the number of bits used to display one color.

All modern computers use 24 bit or 32 bit hardware for color resolution:

* 24 bits =      16,777,216 different "True Colors"
* 32 bits = 4,294,967,296 different "Deep Colors"

The screen.pixelDepth property returns the pixel depth of the screen.

For modern computers, Color Depth and Pixel Depth are equal.

JavaScript Window Location

The window.location object can be used to get the current page address (URL) and to redirect the browser to a new page.

## Window Location

The window.location object can be written without the window prefix.

Some examples:

* window.location.href returns the href (URL) of the current page
* window.location.hostname returns the domain name of the web host
* window.location.pathname returns the path and filename of the current page
* window.location.protocol returns the web protocol used (http: or https:)
* window.location.assign() loads a new document

JavaScript Window History

The window.history object contains the browsers history.

To protect the privacy of the users, there are limitations to how JavaScript can access this object.

* history.back() - same as clicking back in the browser
* history.forward() - same as clicking forward in the browser

JavaScript Window Navigator

The window.navigator object contains information about the visitor's browser.

console.log(navigator.appName) *//Netscape // The appName property returns the application name of the browser:*

console.log(navigator.appCodeName) *// Mozilla*

console.log(navigator.platform)*// Win32*

## Browser Cookies

The cookieEnabled property returns true if cookies are enabled, otherwise false:

console.log(navigator.cookieEnabled) *//true*

Strange enough, "Netscape" is the application name for both IE11, Chrome, Firefox, and Safari.

"Mozilla" is the application code name for both Chrome, Firefox, IE, Safari, and Opera.

console.log(navigator.product) *// Gecko*

Do not rely on this. Most browsers returns "Gecko" as product name !!

The appName property returns the application name of the browser:

The appCodeName property returns the application code name of the browser:

The product property returns the product name of the browser engine:

The appVersion property returns version information about the browser:

The userAgent property returns the user-agent header sent by the browser to the server:

The platform property returns the browser platform (operating system):

The language property returns the browser's language:

console.log(navigator.language) *// en-US*

The onLine property returns true if the browser is online:

console.log(navigator.onLine) *// true*

The javaEnabled() method returns true if [Java](https://www.w3schools.com/java/default.asp) is enabled

console.log(navigator.appName) *//Netscape*

console.log(navigator.appCodeName) *// Mozilla*

console.log(navigator.platform)*// Win32*

console.log(navigator.cookieEnabled) *//true*

console.log(navigator.product) *// Gecko*

console.log(navigator.appVersion)*//5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4324.190 Safari/537.36*

console.log(navigator.userAgent)*//Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4324.190 Safari/537.36*

console.log(navigator.language) *// en-US*

console.log(navigator.onLine) *// true*

console.log(navigator.javaEnabled()) *//false*

JavaScript Cookies

Cookies let you store user information in web pages.

## What are Cookies?

Cookies are data, stored in small text files, on your computer.

When a web server has sent a web page to a browser, the connection is shut down, and the server forgets everything about the user.

Cookies were invented to solve the problem "how to remember information about the user":

* When a user visits a web page, his/her name can be stored in a cookie.
* Next time the user visits the page, the cookie "remembers" his/her name.

Cookies are saved in name-value pairs like:

username = John Doe

When a browser requests a web page from a server, cookies belonging to the page are added to the request. This way the server gets the necessary data to "remember" information about users.

## Create a Cookie with JavaScript

JavaScript can create, read, and delete cookies with the document.cookie property.

document.cookie = "username=John Doe";

You can also add an expiry date (in UTC time). By default, the cookie is deleted when the browser is closed:

document.cookie = "username=John Doe; expires=Sun Mar 07 2021 10:33:50 IST";

With a path parameter, you can tell the browser what path the cookie belongs to. By default, the cookie belongs to the current page.

document.cookie = "username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC; path=/";

## Read a Cookie with JavaScript

document.cookie = "username=John Doe; expires=Thu, 18 Dec 2021 12:00:00 UTC; path=/";

var x = document.cookie;

console.log(x) *//username=John Doe*

document.cookie will return all cookies in one string much like: cookie1=value; cookie2=value; cookie3=value;

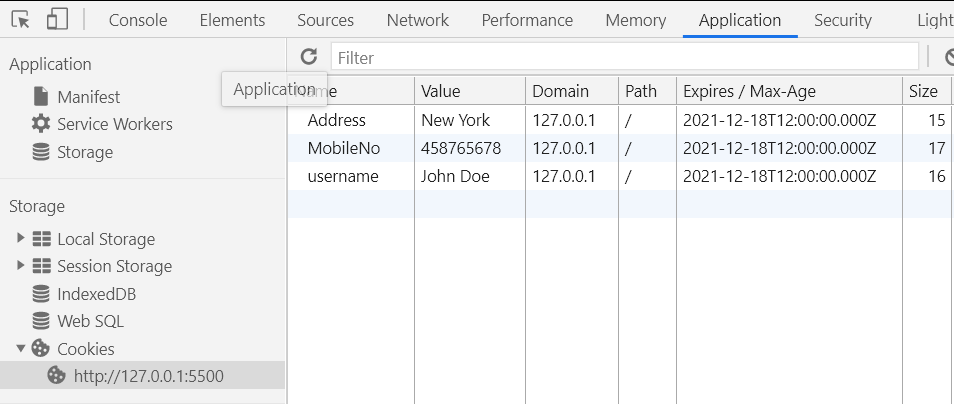
document.cookie = "username=John Doe; expires=Thu, 18 Dec 2021 12:00:00 UTC; path=/";

document.cookie = "Address=New York; expires=Thu, 18 Dec 2021 12:00:00 UTC; path=/";

document.cookie = "MobileNo=458765678; expires=Thu, 18 Dec 2021 12:00:00 UTC; path=/";

var x = document.cookie;

console.log(x) *//username=John Doe; Address=New York; MobileNo=458765678*



## Delete a Cookie with JavaScript

You don't have to specify a cookie value when you delete a cookie.

Just set the expires parameter to a passed date:

You should define the cookie path to ensure that you delete the right cookie.

Some browsers will not let you delete a cookie if you don't specify the path.

## JavaScript Cookie Example

In the example to follow, we will create a cookie that stores the name of a visitor.

The first time a visitor arrives to the web page, he/she will be asked to fill in his/her name. The name is then stored in a cookie.

The next time the visitor arrives at the same page, he/she will get a welcome message.

For the example we will create 3 JavaScript functions:

1. A function to set a cookie value
2. A function to get a cookie value
3. A function to check a cookie value

checkCookie();

function setCookie(cname, cvalue, exdays) {

    var d = new Date();

    d.setTime(d.getTime() + (exdays \* 24 \* 60 \* 60 \* 1000));

    var expires = "expires=" + d.toGMTString();

    document.cookie = cname + "=" + cvalue + ";" + expires + ";path=/";

}

function getCookie(cname) {

    var name = cname + "=";

    var decodedCookie = decodeURIComponent(document.cookie);

    var ca = decodedCookie.split(';');

    for (var i = 0; i < ca.length; i++) {

        var c = ca[i];

        while (c.charAt(0) == ' ') {

            c = c.substring(1);

        }

        if (c.indexOf(name) == 0) {

            return c.substring(name.length, c.length);

        }

    }

    return "";

}

function checkCookie() {

    var user = getCookie("username");

    if (user != "") {

        alert("Welcome again " + user);

    } else {

        user = prompt("Please enter your name:", "");

        if (user != "" && user != null) {

            setCookie("username", user, 30);

        }

    }}

AJAX Introduction

AJAX is a developer's dream, because you can:

* Read data from a web server - after the page has loaded
* Update a web page without reloading the page
* Send data to a web server - in the background

AJAX just uses a combination of:

* A browser built-in XMLHttpRequest object (to request data from a web server)
* JavaScript and HTML DOM (to display or use the data)

AJAX

AJAX is not a programming language.

AJAX is a technique for accessing web servers from a web page.

AJAX stands for Asynchronous JavaScript And XML.

<div *id*="demo">

        <h2>Let AJAX change this text</h2>

        <button *type*="button" *onclick*="loadDoc()">Change Content</button>

</div>

App.js

function loadDoc() {

    var xhttp = new XMLHttpRequest();

    xhttp.onreadystatechange = function () {

        if (this.readyState == 4 && this.status == 200) {

            document.getElementById("demo").innerHTML = this.responseText;

        }

    };

    xhttp.open("GET", "ajax\_info.txt", true);

    xhttp.send();

}

AJAX is a misleading name. AJAX applications might use XML to transport data, but it is equally common to transport data as plain text or JSON text.

AJAX allows web pages to be updated asynchronously by exchanging data with a web server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page.



AJAX - The XMLHttpRequest Object

The keystone of AJAX is the XMLHttpRequest object.

All modern browsers support the XMLHttpRequest object.

The XMLHttpRequest object can be used to exchange data with a web server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page.

## Create an XMLHttpRequest Object

var xhttp = new XMLHttpRequest();

## Access Across Domains

For security reasons, modern browsers do not allow access across domains.

This means that both the web page and the XML file it tries to load, must be located on the same server.

## Modern Browsers (Fetch API)

Modern Browsers can use Fetch API instead of the XMLHttpRequest Object.

The Fetch API interface allows web browser to make HTTP requests to web servers.

If you use the XMLHttpRequest Object, Fetch can do the same in a simpler way.

## XMLHttpRequest Object Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| new XMLHttpRequest() | Creates a new XMLHttpRequest object |
| abort() | Cancels the current request |
| getAllResponseHeaders() | Returns header information |
| getResponseHeader() | Returns specific header information |
| open(*method, url, async, user, psw*) | Specifies the request  *method*: the request type GET or POST *url*: the file location *async*: true (asynchronous) or false (synchronous) *user*: optional user name *psw*: optional password |
| send() | Sends the request to the server Used for GET requests |
| send(*string*) | Sends the request to the server. Used for POST requests |
| setRequestHeader() | Adds a label/value pair to the header to be sent |

## XMLHttpRequest Object Properties

|  |  |
| --- | --- |
| **Property** | **Description** |
| onreadystatechange | Defines a function to be called when the readyState property changes |
| readyState | Holds the status of the XMLHttpRequest. 0: request not initialized 1: server connection established 2: request received 3: processing request 4: request finished and response is ready |
| responseText | Returns the response data as a string |
| responseXML | Returns the response data as XML data |
| status | Returns the status-number of a request 200: "OK" 403: "Forbidden" 404: "Not Found" For a complete list go to the [Http Messages Reference](https://www.w3schools.com/tags/ref_httpmessages.asp) |
| statusText | Returns the status-text (e.g. "OK" or "Not Found") |

AJAX - Send a Request To a Server

The XMLHttpRequest object is used to exchange data with a server.

To send a request to a server, we use the open() and send() methods of the XMLHttpRequest object:

xhttp.open("GET", "ajax\_info.txt", true);  
xhttp.send();

|  |  |
| --- | --- |
| **Method** | **Description** |
| open(*method, url, async*) | Specifies the type of request  *method*: the type of request: GET or POST *url*: the server (file) location *async*: true (asynchronous) or false (synchronous) |
| send() | Sends the request to the server (used for GET) |
| send(*string*) | Sends the request to the server (used for POST) |

Web APIs - Introduction

A Web API is a developer's dream.

* It can extend the functionality of the browser
* It can greatly simplify complex functions
* It can provide easy syntax to complex code

## What is Web API?

API stands for **A**pplication **P**rogramming **I**nterface.

A Web API is an application programming interface for the Web.

A Browser API can extend the functionality of a web browser.

A Server API can extend the functionality of a web server.

## Browser APIs

## All browsers have a set of built-in Web APIs to support complex operations, and to help accessing data.

For example, the Geolocation API can return the coordinates of where the browser is located.

var x = document.getElementById("demo");

function getLocation() {

    if (navigator.geolocation) {

        navigator.geolocation.getCurrentPosition(showPosition);

    } else {

        x.innerHTML = "Geolocation is not supported by this browser.";

    }

}

function showPosition(position) {

    x.innerHTML = "Latitude: " + position.coords.latitude +

        "<br>Longitude: " + position.coords.longitude;

}

getLocation();

## Third Party APIs

Third party APIs are not built into your browser.

To use these APIs, you will have to download the code from the Web.

Examples:

* YouTube API - Allows you to display videos on a web site.
* Twitter API - Allows you to display Tweets on a web site.
* Facebook API - Allows you to display Facebook info on a web site.

Web History API

The Web History API provides easy methods to access the windows.history object.

The window.history object contains the URLs (Web Sites) visited by the user.

## The History back() Method

The back() method loads the previous URL in the windows.history list.

It is the same as clicking the "back arrow" in your browser.

window.history.back();

## The History go() Method

The go() method loads a specific URL from the history list:

window.history.go(-2);

## History Object Properties

|  |  |
| --- | --- |
| **Property** | **Description** |
| [length](https://www.w3schools.com/jsref/prop_his_length.asp) | Returns the number of URLs in the history list |

## History Object Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| [back()](https://www.w3schools.com/jsref/met_his_back.asp) | Loads the previous URL in the history list |
| [forward()](https://www.w3schools.com/jsref/met_his_forward.asp) | Loads the next URL in the history list |
| [go()](https://www.w3schools.com/jsref/met_his_go.asp) | Loads a specific URL from the history list |

Web Storage API

The Web Storage API is a simple syntax for storing and retrieving data in the browser. It is very easy to use:

localStorage.setItem("name", "John Doe");

localStorage.getItem("name");

## The localStorage Object

The localStorage object provides access to a local storage for a particular Web Site. It allows you to store, read, add, modify, and delete data items for that domain.

The data is stored with no expiration date, and will not be deleted when the browser is closed.

The data will be available for days, weeks, and years.

## The sessionStorage Object

The sessionStorage object is identical to the localStorage object.

The difference is that the sessionStorage object stores data for one session.

The data is deleted when the browser is closed.

sessionStorage.setItem("name", "John Doe");

sessionStorage.getItem("name");

# Web Workers API

# A web worker is a JavaScript running in the background, without affecting the performance of the page.

# When executing scripts in an HTML page, the page becomes unresponsive until the script is finished.

## Web Workers Example

The example below creates a simple web worker that count numbers in the background:

## Check Web Worker Support

Before creating a web worker, check whether the user's browser supports it:

## Create a Web Worker File

Now, let's create our web worker in an external JavaScript.

Here, we create a script that counts. The script is stored in the "demo\_workers.js" file:

The important part of the code above is the postMessage() method - which is used to post a message back to the HTML page.

**Note:** Normally web workers are not used for such simple scripts, but for more CPU intensive tasks.

## Create a Web Worker Object

Now that we have the web worker file, we need to call it from an HTML page.

## Terminate a Web Worker

When a web worker object is created, it will continue to listen for messages (even after the external script is finished) until it is terminated.

To terminate a web worker, and free browser/computer resources, use the terminate() method:

## Reuse the Web Worker

If you set the worker variable to undefined, after it has been terminated, you can reuse the code:

## Web Workers and the DOM

Since web workers are in external files, they do not have access to the following JavaScript objects:

* The window object
* The document object
* The parent object

# App.js

var w;

function startWorker() {

    if (typeof (Worker) !== "undefined") {

        if (typeof (w) == "undefined") {

            w = new Worker("demo\_workers.js");

        }

        w.onmessage = function (event) {

            document.getElementById("result").innerHTML = event.data;

        };

    } else {

        document.getElementById("result").innerHTML = "Sorry, your browser does not support Web Workers...";

    }

}

function stopWorker() {

    w.terminate();

    w = undefined;

}

# Demo\_workers.js

var i = 0;

function timedCount() {

  i = i + 1;

  postMessage(i);

  setTimeout("timedCount()",500);

}

timedCount();

# index.html

<p>Count numbers: <output *id*="result"></output></p>

    <button *onclick*="startWorker()">Start Worker</button>

    <button *onclick*="stopWorker()">Stop Worker</button>

JavaScript Fetch API

The Fetch API interface allows web browser to make HTTP requests to web servers.

If you use the XMLHttpRequest Object, Fetch can do the same in a simpler way.

# Fetch is based on async and await

*async* function getText(file) {

    let myObject = await fetch(file);

    let myText = await myObject.text();

    console.log(myText);

}

getText('https://api.npoint.io/b102ac3f22d42bb6763e');

fetch('https://api.npoint.io/b102ac3f22d42bb6763e')

    .then(x => x.text())

    .then(y => console.log(y));