Javascript as a client-side programming language

Javascript can run as a client-side or server-side language

Client-side: javascript is embedded in html page (in <script>…</script>), which is run in a web browser

Server-side: Node (based on Google’s V8 engine) is the compiler

# Javascript language

# Run Javascript with HTML

## Run Javascript in browser or Node

## Where to put Javascript

In HTML, JavaScript code is inserted between <script> and </script> tags.

<script>  
document.getElementById("demo").innerHTML = "My First JavaScript";  
</script>

**You can place any number of scripts in an HTML document.**

Scripts can be placed in the <body>, or in the <head> section of an HTML page, or in both.

Scripts can also be placed in external files of extension .js:

External file: myScript.js

function myFunction() {  
  document.getElementById("demo").innerHTML = "Paragraph changed.";  
}

External scripts are practical when the same code is used in many different web pages.

To use an external script, put the name of the script file in the src (source) attribute of a <script> tag:

Example: Instead of <script> …code… </script>, put the content …code… in an external file and add the following line

<script src="myScript.js"></script>

You can place an external script reference in <head> or <body> as you like.

The script will behave as if it was located exactly where the <script> tag is located.

# JavaScript Interact with users

## alert() to show a message, prompt() for input, confirm() for yes/no confirmation

As we’ll be using the browser as our demo environment, let’s see a couple of functions to interact with the user: alert, prompt and confirm.

### [alert](https://javascript.info/alert-prompt-confirm" \l "alert)

It shows a message and waits for the user to press “OK”.

For example:

alert("Hello");

The mini-window with the message is called a modal window. The word “modal” means that the visitor can’t interact with the rest of the page, press other buttons, etc, until they have dealt with the window. In this case – until they press “OK”.

### [prompt](https://javascript.info/alert-prompt-confirm" \l "prompt)

The function prompt accepts two arguments:

result = prompt(title, [default]);

It shows a modal window with a text message, an input field for the visitor, and the buttons OK/Cancel.

**Title** The text to show the visitor.

**Default** An optional second parameter, the initial value for the input field.

**The square brackets in syntax [...]**

The square brackets around default in the syntax above denote that the parameter is optional, not required.

The visitor can type something in the prompt input field and press OK. Then we get that text in the result. Or they can cancel the input by pressing Cancel or hitting the Esc key, then we get null as the result.

The call to prompt returns the text from the input field or null if the input was canceled.

For instance:

let age = prompt('How old are you?', 100);

alert(`You are ${age} years old!`); // You are 100 years old!

**In IE: always supply a default**

The second parameter is optional, but if we don’t supply it, Internet Explorer will insert the text "undefined" into the prompt.

Run this code in Internet Explorer to see:

let test = prompt("Test");

So, for prompts to look good in IE, we recommend always providing the second argument:

let test = prompt("Test", ''); // <-- for IE

### [confirm](https://javascript.info/alert-prompt-confirm" \l "confirm)

The syntax:

result = confirm(question);

The function confirm shows a modal window with a question and two buttons: OK and Cancel.

The result is true if OK is pressed and false otherwise.

For example:

let isBoss = confirm("Are you the boss?");

alert( isBoss ); // true if OK is pressed

## JavaScript Output

JavaScript can "display" data in different ways:

* Writing into an HTML element, using innerHTML.
* Writing into the HTML output using document.write().
* Writing into an alert box, using window.alert().
* Writing into the browser console, using console.log().

### Using innerHTML

To access an HTML element, JavaScript can use the document.getElementById(id) method.

The id attribute defines the HTML element. The innerHTML property defines the HTML content:

Example

<!DOCTYPE html>  
<html>  
<body>  
  
<h1>My First Web Page</h1>  
<p>My First Paragraph</p>  
  
<p id="demo"></p>  
  
<script>  
document.getElementById("demo").innerHTML = 5 + 6;  
</script>  
  
</body>  
</html>

Changing the innerHTML property of an HTML element is a common way to display data in HTML.

### Using document.write()

For testing purposes, it is convenient to use document.write():

Example

<!DOCTYPE html>  
<html>  
<body>  
  
<h1>My First Web Page</h1>  
<p>My first paragraph.</p>  
  
<script>  
document.write(5 + 6);  
</script>  
  
</body>  
</html>

Using document.write() **after** an HTML document is loaded, will **delete all existing HTML**:

The following Example is different from the above in that document.write() is called after the document is loaded.

<!DOCTYPE html>  
<html>  
<body>  
  
<h1>My First Web Page</h1>  
<p>My first paragraph.</p>  
  
<button type="button" onclick="document.write(5 + 6)">Try it</button>  
  
</body>  
</html>

The document.write() method should only be used for testing.

### Using window.alert()

You can use an alert box to display data:

Example

<!DOCTYPE html>  
<html>  
<body>  
  
<h1>My First Web Page</h1>  
<p>My first paragraph.</p>  
  
<script>  
window.alert(5 + 6);  
</script>  
  
</body>  
</html>

You can skip the window keyword.

In JavaScript, the window object is the global scope object, that means that variables, properties, and methods by default belong to the window object. This also means that specifying the window keyword is optional:

Example

<!DOCTYPE html>  
<html>  
<body>  
  
<h1>My First Web Page</h1>  
<p>My first paragraph.</p>  
  
<script>  
alert(5 + 6);  
</script>  
  
</body>  
</html>

### Using console.log()

For debugging purposes, you can call the console.log() method in the browser to display data.

Example

<!DOCTYPE html>  
<html>  
<body>  
  
<script>  
console.log(5 + 6);  
</script>  
  
</body>  
</html>

### JavaScript Print

JavaScript does not have any print object or print methods.

You cannot access output devices from JavaScript.

The only exception is that you can call the window.print() method in the browser to print the content of the current window.

Example

<!DOCTYPE html>  
<html>  
<body>  
  
<button onclick="window.print()">Print this page</button>  
  
</body>  
</html>

# JavaScript Syntax

Javascript statements are executed, one by one, in the same order as they are written.

Semicolons separate JavaScript statements.

JavaScript ignores multiple spaces. You can add white space to your script to make it more readable.

Two types of values in Javascript: Fixed values (Literals) and Variable values (variables)

Literals:

* **Numbers** are written with or without decimals
* **Strings** are text, written within double or single quotes

Variables: JavaScript uses the var keyword to **declare** variables. An **equal sign** is used to **assign values** to variables.

JavaScript Expressions, e.g. 5\*10, x+3, without assignment will be displayed.

JavaScript Identifiers (names) are used to name variables, functions: the first character must be a letter, or an underscore (\_), or a dollar sign ($).

JavaScript is case-sensitive and uses lower camel case.

# Data types

## Primitive (boolean, number, string, undefined, null, symbol) and Object

All values belong to one of 8 types:

|  |  |
| --- | --- |
| **Type** | **Values of the Type** |
| Undefined | Only one value: undefined. Means “I don’t know,” “I don’t care”, or “None of your business.” |
| Null | Only one value: null. Means “no value.” |
| Boolean | Only two values: true and false. |
| Number | The IEEE 754 64-bit floating point values. Values that are integers can be expressed in binary, octal, decimal, or hex; non-integers must be expressed in decimal. Examples:   * 8 * 7.23342 * 6.02e23 * 0xff3e * 0b11010100001010 * 0o237 * Infinity * NaN |
| BigInt | Arbitrary-precision integers. Needed because the Number type can not represent most integers with a magnitude above 9007199254740992. Examples:   * 3n * 2098321521257182187525313919187155317815353517831735173173551735173n |
| String | Immutable sequences of zero or more UTF-16 code units. You can delimit them with apostrophes, quotation marks, or backticks. Examples:   * "hello" * "She said 'I don’t think so 😎'... (╯°□°）╯︵ ┻━┻)" * 'x = "4"' * "abc\tdef\"\r\nghi\n🏄‍♀️🏀\n" * "Olé" * "Ol\xe9" * 'Will I?\u043d\u0435\u0442\u263a' * `The sum of ${x} and ${y} is probably ${x + y}`   Only backtick-delimited literals can span lines and support interpolation. |
| Symbol | Unique things. Every time you create a symbol, you get a new thing, different from all other symbols. This is not necessarily true of strings. Examples:   * Symbol() * Symbol('dog') * Symbol('dog') // different from the one above |
| Object | Everything that isn’t one of the above types. Examples:   * {} * {latitude: 74.2, longitude: -153.11} * [true, true, {last: false, value: 'okay'}, [0, 0, 2]] * new Set([5, 1, 2]) * new Date(2000, 12, 31) * (x, y) => x \* x + y \* y * /Boo+m!?/gi   Certain kinds of objects, such as arrays, functions and regular expressions have special syntactic forms, but to JavaScript they are just considered to have the type Object. |

## Primitve are immutable, Objects are mutable

A **mutable object** is an object whose state can be modified after it is created.

**Immutables** are the objects whose state cannot be changed once the object is created.

In [JavaScript](https://developer.mozilla.org/en-US/docs/Glossary/JavaScript), only [objects](https://developer.mozilla.org/en-US/docs/Glossary/Object) and [arrays](https://developer.mozilla.org/en-US/docs/Glossary/Array) are mutable. **Strings and Numbers** are **Immutable**. Lets understand this with an example:

var immutableString = "Hello";

// In the above code, a new object with string value is created.

immutableString = immutableString + "World";

// We are now appending "World" to the existing value.

On appending the "immutableString" with a string value, following events occur:

1. Existing value of "immutableString" is retrieved
2. "World" is appended to the existing value of "immutableString"
3. The resultant value is then allocated to a new block of memory
4. "immutableString" object now points to the newly created memory space
5. Previously created memory space is now available for garbage collection.

## Objects

An instance of Object data type is called an object, which is a collection of related variables and functions.

### Create an object

There are two ways to create an object:

* Create an object literal by writing out the object content:

const person = {

name: ['Bob', 'Smith'],

age: 32,

gender: 'male',

interests: ['music', 'skiing'],

bio: function() {

alert(this.name[0] + ' ' + this.name[1] + ' is ' + this.age + ' years old. He likes ' + this.interests[0] + ' and ' + this.interests[1] + '.');

},

greeting: function() {

alert('Hi! I\'m ' + this.name[0] + '.');

}

};

* Instantiate an object of Object class:

let person = new Object();

person.name = "Nicholas";

person.age = 29;

It’s also possible to create an object with only the default properties and methods using object literal notation by leaving the space between the curly braces empty, such as this for the above example:

let person = {}; // same as new Object()

person.name = "Nicholas";

person.age = 29;

Note: A member of an object can be another object.

For example, try changing the name member from

name: ['Bob', 'Smith'],

to

name : {

first: 'Bob',

last: 'Smith'

},

Here we are effectively creating a **sub-namespace**. This sounds complex, but really it's not — to access these items you just need to chain the extra step onto the end with another dot. Try these in the JS console:

person.name.first

person.name.last

### Get/Set data of an object by dot or bracket notation

To get data of an object:

Dot

person.name

person.name[0]

person.age

person.interests[1]

person.bio()

person.greeting()

Bracket notation

person['age']

To set data of an object:

person.age = 45;

person['name']['last'] = 'Cratchit';

Setting members doesn't just stop at updating the values of existing properties and methods; you can also create completely new members. Try these in the JS console:

person['eyes'] = 'hazel';

person.farewell = function() { alert("Bye everybody!"); }

**Bracket notation can be used to create dynamically new data members**

For example: create a new data member whose name, value is given dynamically by myDataName, myDataValue:

let myDataName = nameInput.value;

let myDataValue = nameValue.value;

We could then add this new member name and value to the person object like this:

person[myDataName] = myDataValue;

### this keyword refers to the current object

The this keyword refers to the current object the code is being written inside — so in the example of person object above, this is equivalent to person.

greeting: function() {

alert('Hi! I\'m ' + this.name.first + '.');

}

### Global object

The global object provides variables and functions that are available anywhere. By default, those that are built into the language or the environment.

In a browser it is named window, for Node.js it is global, for other environments it may have another name.

Recently, globalThis was added to the language, as a standardized name for a global object, that should be supported across all environments. It’s supported in all major browsers.

We’ll use window here, assuming that our environment is a browser. If your script may run in other environments, it’s better to use globalThis instead.

All properties of the global object can be accessed directly:

alert("Hello");

// is the same as

window.alert("Hello");

In a browser, global functions and variables declared with var (not let/const!) become the property of the global object:

var gVar = 5;

alert(window.gVar); // 5 (became a property of the global object)

The same effect have function declarations (statements with function keyword in the main code flow, not function expressions).

Please don’t rely on that! This behavior exists for compatibility reasons. Modern scripts use [JavaScript modules](https://javascript.info/modules) where such thing doesn’t happen.

If we used let instead, such thing wouldn’t happen:

let gLet = 5;

alert(window.gLet); // undefined (doesn't become a property of the global object)

If a value is so important that you’d like to make it available globally, write it directly as a property:

// make current user information global, to let all scripts access it

window.currentUser = {

name: "John"

};

// somewhere else in code

alert(currentUser.name); // John

// or, if we have a local variable with the name "currentUser"

// get it from window explicitly (safe!)

alert(window.currentUser.name); // John

That said, using global variables is generally discouraged. There should be as few global variables as possible. The code design where a function gets “input” variables and produces certain “outcome” is clearer, less prone to errors and easier to test than if it uses outer or global variables.

## Array

An array is a list of elements that are implemented by a sequence of **consecutive** memory unit

### Create an array and access its element

#### Create an array like a literal or an instance of Array class

* Literal:

var cars = ["Saab", "Volvo", "BMW"];

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

* Create an instance of Array class

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

It’s better to create an array as a literal than an instance of Array class

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

but

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

#### Set/get elements of array

Access the whole array by using the array’s name

var cars = ["Saab", "Volvo", "BMW"];  
document.getElementById("demo").innerHTML = cars;

Access an element through its index (which starts out by 0):

var cars = ["Saab", "Volvo", "BMW"];  
document.getElementById("demo").innerHTML = cars[0];

cars[0] = "Opel";

var x = cars.length;   // The length property returns the number of elements  
var y = cars.sort();   // The sort() method sorts arrays

New element can also be added to an array using the length property:

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits[fruits.length] = "Lemon";    // adds a new element (Lemon) to fruits

### Array vs Object

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

Arrays use **numbers** to access its "elements" while Objects use **names**. So in the following example, for array: person[0] returns “John”, for object: person.firstName or person[‘firstName’] returns John

var person = ["John", "Doe", 46];

var person = {firstName:"John", lastName:"Doe", age:46};

### push() pop() splice()

#### push() or pop() the last element

The pop() method removes the last element from an array:

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.pop();              // Removes the last element ("Mango") from fruits

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

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.push("Kiwi");       //  Adds a new element ("Kiwi") to fruits

#### Add or remove any element with splice()

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

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.splice(2, 1, "Lemon", "Kiwi");

Original Array:  
Banana,Orange,Apple,Mango

New Array:  
Banana,Orange,Lemon,Kiwi,Apple,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 a new array.

The splice() can be used 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.

### sort()

The sort() function, by default sorts values as **strings** in the alphabetical order. The reverse() method reverses that order

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.sort();        // Sorts the elements of fruits so Apple comes before “Banana”

You can provide a compare function to sort your array according to your own order.

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.

For example, to sort an array of numbers ascendingly:

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

To sort an array in a random order:

var points = [40, 100, 1, 5, 25, 10];  
points.sort(function(a, b){return 0.5 - Math.random()});

## Operator == vs ===

The difference between == and === is that:

* == **converts** the variable values to the **same** type before performing comparison. This is called [type coercion](https://developer.mozilla.org/en-US/docs/Glossary/Type_coercion).
* === does **not** do any type conversion (coercion) and returns true only **if** both values **and** types are identical for the two variables being compared.

var one = 1;

var one\_again = 1;

var one\_string = "1"; // note: this is string

console.log(one == one\_again); // true

console.log(one === one\_again); // true

console.log(one == one\_string); // true. See below for explanation.

console.log(one === one\_string); // false. See below for explanation.

= and === have their counterparts when it comes to checking for inequality:

* !=: Converts values if variables are different types before checking for inequality
* !==: Checks both type and value for the two variables being compared

var one = 1;

var one\_again = 1;

var one\_string = "1"; // note: this is a string

console.log(one != one\_again); // false

console.log(one != one\_string); // false

console.log(one !== one\_string);// true. Types are different

## typeof returns the type of a variable

The typeof operator returns a string indicating the type of the following operand.

typeof operand

typeof(operand)

Example:

typeof 37 === 'number';

typeof '' === 'string';

typeof Infinity === 'number';

typeof NaN === 'number'; // Despite being "Not-A-Number"

// Objects

typeof {a: 1} === 'object';

// use Array.isArray or Object.prototype.toString.call

// to differentiate regular objects from arrays

typeof [1, 2, 4] === 'object';

typeof new Date() === 'object';

typeof undeclaredVariable === 'undefined';

// Functions

typeof function() {} === 'function';

typeof class C {} === 'function';

typeof Math.sin === 'function';

# Functions and function type

## function expression and function type

Javascript’s function declaration requires “function” keyword, but no return type, no argument types.

Function declaration can be used as a value in an expression, so comes the name function expression.

<script>

    function factorial(n){

        if(n <= 1) return 1;

        else return n\*factorial(n-1);

    }

    // Show the value of factorial, factorial(4)

    console.log('The source code of factorial function is: \n' + factorial);

    console.log('The value of factorial function at 4 is: ' + factorial(4));

    // Now change the value of factorial

    factorial = function negativeOne(){return -1;}

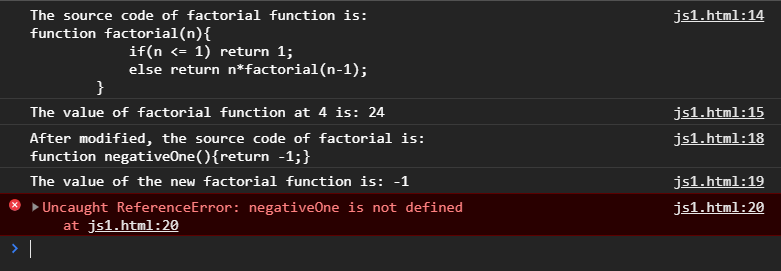
    console.log('After modified, the source code of factorial is: \n' + factorial);

    console.log('The value of the new factorial function is: ' + factorial());

    console.log('The value of negativeOne function is ' + negativeOne()); // Doesn’t compile

</script>

Output:



When you declare a function (but don’t assign it explicitly to any variable), JS automatically declares a variable whose type is ‘function’ (a kind of Object), whose name is the function name, and whose value is the function source code. When you call this function variable, you get the source code of the function. If you add “(argument)” to the function variable, you get the value of function at argument.

When you assign a function expression (i.e. the function declaration) to a variable (so the variable is of ‘function’ type), you cannot call the function through the function name, but only through the variable name so the function name becomes optional. The function name is still needed in the case you want to do recursion, like calculating factorial.

<script>

    var x = function factorial(n){

        if(n <= 1) return 1;

        else return n\*factorial(n-1);

    }

    // Show the value of factorial, factorial(4) through x

    console.log('The source code of factorial function is: \n' + x);

    console.log('The value of factorial function at 4 is: ' + x(4));

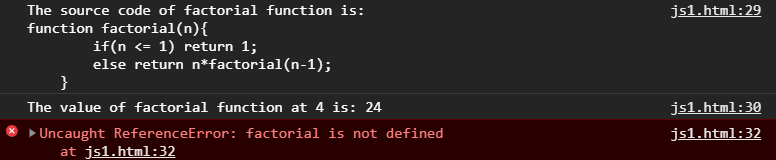
    // Show the value of factorial, factorial(4) through factorial --> error

    console.log('The source code of factorial function is: \n' + factorial); // error

    console.log('The value of factorial function at 4 is: ' + factorial(4)); // error

</script>

Output:



Function expressions are sometimes defined and immediately invoked:

let fourFactorial = (function factorial(n){

    if(n <= 1) return 1;

    else return n\*factorial(n-1);

}(4));

// though 'factorial' name is optional, it's needed here for recursion

Function declaration statements are “hoisted” to the top of the enclosing script, function, or block so that functions defined in this way may be invoked from code that appears before the definition. In contrast, for a function that is defined in a function expression, you can only use the function after the function expression.

## Callback functions

Function expression can be passed as an argument into another function. Firstly, function *arg* is passed into function *Func* as an argument and then while being executed, Func calls arg back, and hence function arg is called a “callback” function.

Example: We’ll write a function ask(question, yes, no) that ask the question and, depending on the user’s answer, call yes() or no():

function ask(question, yes, no) {

if (confirm(question)) yes()

else no();

}

function showOk() {

alert( "You agreed." );

}

function showCancel() {

alert( "You canceled the execution." );

}

// usage: functions showOk, showCancel are passed as arguments to ask

ask("Do you agree?", showOk, showCancel);

The “callback” term means we pass a function and expect it to be “called back” later if necessary. In our case, showOk becomes the callback for “yes” answer, and showCancel for “no” answer.

We can use Function Expressions to write the same function much shorter:

function ask(question, yes, no) {

if (confirm(question)) yes()

else no();

}

ask(

"Do you agree?",

function() { alert("You agreed."); },

function() { alert("You canceled the execution."); }

);

## Arrow functions

Arrow functions are a style of writing functions shorter.

let func = function(arg1, arg2, ...argN) {

return expression;

};

Can be shortened as:

let func = (arg1, arg2, ...argN) => expression

Example:

let sum = (a, b) => a + b;

/\* This arrow function is a shorter form of:

let sum = function(a, b) { return a + b;};

\*/

alert( sum(1, 2) ); // 3

When the function is complex, use { }

let sum = (a, b) => { // the curly brace opens a multiline function

let result = a + b;

return result; // if we use curly braces, then we need an explicit "return"

};

alert( sum(1, 2) ); // 3

# Browser environment

## Window object

Browser window is represented by an object called “window” that provide methods for JS code to control. For example we use “window” object to show the window height:

alert(window.innerHeight); // inner window height

This “window” object is a global object, so everything belongs to it, for example when we define a function it becomes a method of “window”

function sayHi() {

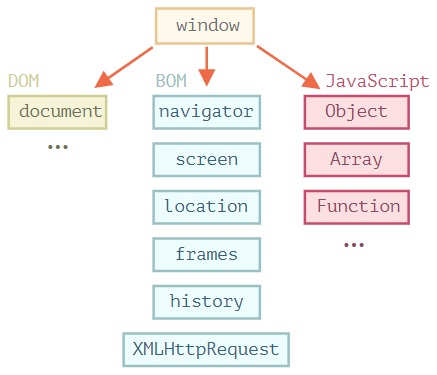
alert("Hello");

}

// global functions are methods of the global object:

window.sayHi();

This window object contains DOM and BOM



### [DOM (Document Object Model)](https://javascript.info/browser-environment" \l "dom-document-object-model)

Document Object Model, or DOM for short, represents all page content as objects that can be modified.

The document object is the main “entry point” to the page. We can change or create anything on the page using it.

For instance:

// change the background color to red

document.body.style.background = "red";

// change it back after 1 second

setTimeout(() => document.body.style.background = "", 1000);

### [BOM (Browser Object Model)](https://javascript.info/browser-environment" \l "bom-browser-object-model)

The Browser Object Model (BOM) represents additional objects provided by the browser (host environment) for working with everything except the document.

For instance:

* The [navigator](https://developer.mozilla.org/en-US/docs/Web/API/Window/navigator) object provides background information about the browser and the operating system. There are many properties, but the two most widely known are: navigator.userAgent – about the current browser, and navigator.platform – about the platform (can help to differ between Windows/Linux/Mac etc).
* The [location](https://developer.mozilla.org/en-US/docs/Web/API/Window/location) object allows us to read the current URL and can redirect the browser to a new one.

Here’s how we can use the location object:

alert(location.href); // shows current URL

if (confirm("Go to Wikipedia?")) {

location.href = "https://wikipedia.org"; // redirect the browser to another URL

}

Functions alert/confirm/prompt are also a part of BOM: they are directly not related to the document, but represent pure browser methods of communicating with the user.

## DOM

The Document Object Model (DOM) model each HTML element as an object and the whole HTML page as a tree of objects so that Javascript can access and change those objects dynamically.

For example, document.body is the object representing the <body> tag and the following code will make the <body> red for 3 seconds:

document.body.style.background = 'red'; // make the background red

setTimeout(() => document.body.style.background = '', 3000); // return back

### HTML tree: document, tag nodes, text nodes, comment nodes

A HTML page is structured as a tree:

* Every HTML tag is an **element** node, the root node is <html>, which is represented by ‘**document**’ object.
* Nested tags are “children” of the enclosing tags, so <head>, <body> are children of <html>.
* The text inside elements forms leaf nodes (i.e. don’t have any children). This leaf node contains only one string and is called a **text** node.
* Spaces and newlines are totally valid characters, like letters and digits, so they form text nodes.
* **Comments** are nodes, too. Though they are not shown, Javascript can access them.

(so 4 types: *document* node (root), *element* nodes (tag), *text* nodes (text) and *comment* nodes)

Note: spaces and newlines before <head> are ignored. Everything after </body> will be moved into <body>.

Example: The DOM of the following HTML page

<!DOCTYPE HTML>

<html>

<head>

<title>About elk</title>

</head>

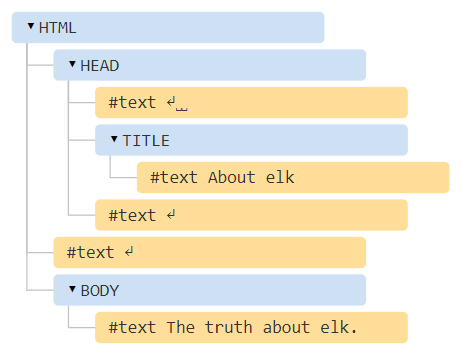
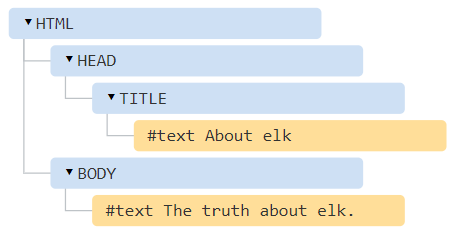
<body>

The truth about elk.

</body>

</html>

is:



The <head> tag contains some spaces, line feed before/after <title>, <body> and they become text nodes. To avoid those, the HTML page should be like:

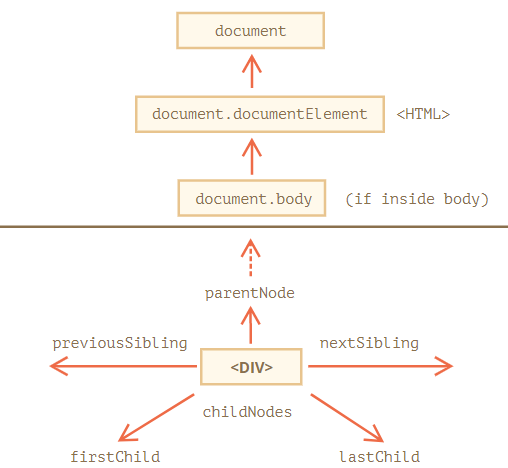
<!DOCTYPE HTML>

<html><head><title>About elk</title></head><body>The truth about elk.</body></html>

### Navigate to the desired DOM object

Each node in the HTML tree above is represented by an object.

All operations on the DOM start with the document object. That’s the main “entry point” to DOM. From it we can access any node by parentNode, nextSibling, childNodes[i], etc.



#### On top: documentElement and body

The topmost tree nodes are available directly as document properties:

**<html> = document.documentElement**

**<body> = document.body**

**<head> = document.head**

To access elements in the HTML page, one often start with document.body and use childNodes, nextSibling, etc to browse the desired element.

**There’s a catch: document.body can be null**

A script cannot access an element that doesn’t exist at the moment of running. In particular, if a script is inside <head>, then document.body is unavailable, because the browser did not read it yet.

<html>

<head>

<script>

alert( "From HEAD: " + document.body ); // null, there's no <body> yet

</script>

</head>

<body>

<script>

alert( "From BODY: " + document.body ); // HTMLBodyElement, now it exists

</script>

</body>

</html>

#### Children: childNodes, firstChild, lastChild

Note: a child is an immediate descendant.

**childNodes is a collection containing all child nodes, including text nodes. One can access a child through an index of childNodes, e.g.** document.body.childNodes[i]

Note that childNodes is not an array; rather, it’s a Javascript iterable so we cannot use methods of Array, but we can use for..of to iterate over it:

for (let node of document.body.childNodes) {

alert(node); // shows all nodes from the collection

}

Properties**firstChild**and**lastChild**give fast access to the first and last children.

elem.childNodes[0] === elem.firstChild

elem.childNodes[elem.childNodes.length - 1] === elem.lastChild

**firstChild**and**lastChild**can be text nodes, comment nodes so if you want only element nodes, use:

**firstElementChild**and**lastElementChild**

Example:

<html>

<body>

<div>Users:</div>

<ul>

<li>John</li>

<li>Pete</li>

</ul>

</body>

</html>

The <div> DOM node:

document.body.firstElementChild

// or

document.body.children[0]

// or (the first node is space, so we take 2nd)

document.body.childNodes[1]

The <ul> DOM node:

document.body.lastElementChild

// or

document.body.children[1]

The second <li> (with Pete):

// get <ul>, and then get its last element child

document.body.lastElementChild.lastElementChild

**DOM collections are read-only** so we can’t replace a child by something by assigning childNodes[i]=...

**DOM collections are live** they reflect the current state of DOM.

#### Siblings and the parent

Siblings are nodes that are children of the same parent.

For instance, here <head> and <body> are siblings:

<html>

<head>...</head><body>...</body>

</html>

* <body> is said to be the “next” or “right” sibling of <head>,
* <head> is said to be the “previous” or “left” sibling of <body>.

The next sibling is in nextSibling property, and the previous one – in previousSibling.

The parent is available as parentNode.

For example:

// parent of <body> is <html>

alert( document.body.parentNode === document.documentElement ); // true

// after <head> goes <body>

alert( document.head.nextSibling ); // HTMLBodyElement

// before <body> goes <head>

alert( document.body.previousSibling ); // HTMLHeadElement

#### Element-only navigation

Navigation properties listed above refer to all nodes. For instance, in childNodes we can see both text nodes, element nodes, and even comment nodes if there exist. If we want only element nodes (not text or comment nodes) then just add Element word to properties’ names in the above section:

* children – only those children that are element nodes.
* firstElementChild, lastElementChild – first and last element children.
* previousElementSibling, nextElementSibling – neighbor elements.
* parentElement – parent element. Except the case of  document.documentElement, parentElement and parent give the same result.

#### Tables

**The <table>** element supports (in addition to the given above) these properties:

* table.rows – the collection of <tr> elements of the table.
* table.caption/tHead/tFoot – references to elements <caption>, <thead>, <tfoot>.
* table.tBodies – the collection of <tbody> elements (can be many according to the standard, but there will always be at least one – even if it is not in the source HTML, the browser will put it in the DOM).

**<thead>, <tfoot>, <tbody>** elements provide the rows property:

* tbody.rows – the collection of <tr> inside.

**<tr>:**

* tr.cells – the collection of <td> and <th> cells inside the given <tr>.
* tr.sectionRowIndex – the position (index) of the given <tr> inside the enclosing <thead>/<tbody>/<tfoot>.
* tr.rowIndex – the number of the <tr> in the table as a whole (including all table rows).

**<td> and <th>:**

* td.cellIndex – the number of the cell inside the enclosing <tr>.

An example of usage:

<table id="table">

<tr>

<td>one</td><td>two</td>

</tr>

<tr>

<td>three</td><td>four</td>

</tr>

</table>

<script>

// get td with "two" (first row, second column)

let td = table.rows[0].cells[1];

td.style.backgroundColor = "red"; // highlight it

</script>

### Searching: getElement\*, querySelector\*

DOM navigation properties are great when elements are close to each other. What if they are not? How to get an arbitrary element of the page?

#### document.getElementById or just id

If an element has the id attribute, we can get the element by:

- using the method document.getElementById(id), no matter where it is.

For instance:

<div id="elem">

<div id="elem-content">Element</div>

</div>

<script>

// get the element

let elem = document.getElementById('elem');

// make its background red

elem.style.background = 'red';

</script>

* call directly a global variable named by id that references the element:

<div id="elem">

<div id="elem-content">Element</div>

</div>

<script>

// elem is a reference to DOM-element with id="elem"

elem.style.background = 'red';

// id="elem-content" has a hyphen inside, so it can't be a variable name

// ...but we can access it using square brackets: window['elem-content']

</script>

Due to some reason, the document.getElementById(id), is preferred over calling the element as a global variable.

#### querySelectorAll

The most versatile method, elem.querySelectorAll(css) returns all elements inside elem matching the given CSS selector.

Here we look for all <li> elements that are last children:

<ul>

<li>The</li>

<li>test</li>

</ul>

<ul>

<li>has</li>

<li>passed</li>

</ul>

<script>

let elements = document.querySelectorAll('ul > li:last-child');

for (let elem of elements) {

alert(elem.innerHTML); // "test", "passed"

}

</script>

This method is indeed powerful, because any CSS selector can be used.

**Can use pseudo-classes as well**

Pseudo-classes in the CSS selector like :hover and :active are also supported. For instance, document.querySelectorAll(':hover') will return the collection with elements that the pointer is over now (in nesting order: from the outermost <html> to the most nested one).

#### querySelector

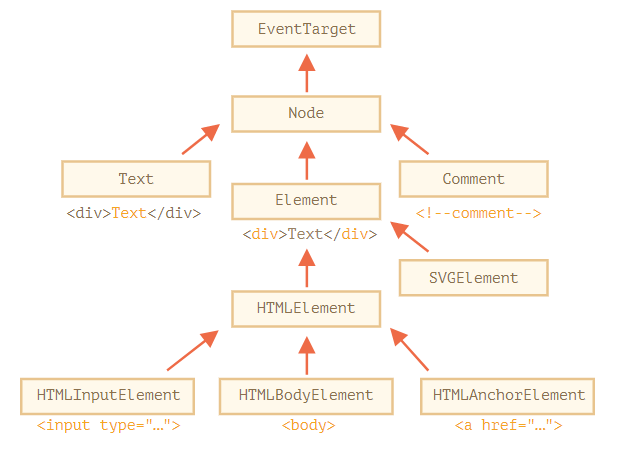
The call to elem.querySelector(css) returns the first element for the given CSS selector.

In other words, the result is the same as elem.querySelectorAll(css)[0], but the latter is looking for all elements and picking one, while elem.querySelector just looks for one. So it’s faster and also shorter to write.

### Node properties: type, name and contents

How node classes inherit properties from the higher

Each DOM object (or node) is an instance of a corresponding built-in Javascript class. For example, DIV object of element node type is an instance of Element class. Those built-in Javascript classes have inheritance hierarchy as:



#### nodeType and nodeName

The nodeType property provides one more, “old-fashioned” way to get the “type” of a DOM node.

It has a numeric value:

* elem.nodeType == 1 for element nodes,
* elem.nodeType == 3 for text nodes,
* elem.nodeType == 9 for the document object,
* there are few other values in [the specification](https://dom.spec.whatwg.org/#node).

Given a DOM node, we can read its tag name from nodeName (or tagName for element nodes) properties.

alert( document.body.nodeName ); // BODY

#### element node’s content: innerHTML and outerHTML

The [innerHTML](https://w3c.github.io/DOM-Parsing/" \l "the-innerhtml-mixin) property allows to get the HTML inside an element node, as a string.

We can also modify it. So it’s one of the most powerful ways to change the page.

The example shows the contents of document.body and then replaces it completely:

<body>

<p>A paragraph</p>

<div>A div</div>

<script>

alert( document.body.innerHTML ); // read the current contents

document.body.innerHTML = 'The new BODY!'; // replace it

</script>

</body>

**Note: Scripts don’t execute**

If innerHTML inserts a <script> tag into the document – it becomes a part of HTML, but doesn’t execute.

The outerHTML property contains the full HTML of the element. That’s like innerHTML plus the element itself.

Here’s an example:

<div id="elem">Hello <b>World</b></div>

<script>

alert(elem.outerHTML); // <div id="elem">Hello <b>World</b></div>

</script>

Unlike innerHTML, writing to outerHTML does not change the element. Instead, it replaces it in the DOM.

#### text node’s content: nodeValue/data

The innerHTML is only valid for element nodes. For text nodes, use: nodeValue and data properties.

An example of reading the content of a text node and a comment:

<body>

Hello

<!-- Comment -->

<script>

let text = document.body.firstChild;

alert(text.data); // Hello

let comment = text.nextSibling;

alert(comment.data); // Comment

</script>

</body>

For text nodes we can imagine a reason to read or modify them, but why comments?

Sometimes developers embed information or template instructions into HTML in them, like this:

<!-- if isAdmin -->

<div>Welcome, Admin!</div>

<!-- /if -->

…Then JavaScript can read it from data property and process embedded instructions.

#### pure text: textContent

The textContent provides access to the text inside the element: only text, minus all <tags>.

**Writing to textContent ensure a “safe way” to write text.**

Example: we have an arbitrary string, for instance entered by a user, and want to show it.

<div id="elem1"></div>

<div id="elem2"></div>

<script>

let name = prompt("What's your name?", "<b>Winnie-the-Pooh!</b>");

elem1.innerHTML = name;

elem2.textContent = name;

</script>

1. The first <div> gets the name “as HTML”: all tags become tags, so we see the bold name.
2. The second <div> gets the name “as text”, so we literally see <b>Winnie-the-Pooh!</b>.

In most cases, we expect the text from a user, and want to treat it as text. We don’t want unexpected HTML in our site. An assignment to textContent does exactly that.

## Events

### Events and event handlers

*An event* is a signal that something has happened. All DOM nodes generate such signals (but events are not limited to DOM).

**Most useful DOM events**

Mouse events:

* click – when the mouse clicks on an element (touchscreen devices generate it on a tap).
* contextmenu – when the mouse right-clicks on an element.
* mouseover / mouseout – when the mouse cursor comes over / leaves an element.
* mousedown / mouseup – when the mouse button is pressed / released over an element.
* mousemove – when the mouse is moved.

Keyboard events:

* keydown and keyup – when a keyboard key is pressed and released.

Form element events:

* submit – when the visitor submits a <form>.
* focus – when the visitor focuses on an element, e.g. on an <input>.

Document events:

* DOMContentLoaded – when the HTML is loaded and processed, DOM is fully built.

CSS events:

* transitionend – when a CSS-animation finishes.

**Event handler**

To react on events we can assign a handler – a function that runs in case of an event.

Handlers are a way to run JavaScript code in case of user actions.

There are 3 ways to assign a handler, through: HTML-attribute, DOM property, addEventListener method.

### Set a handler by HTML attribute: on<event>

A handler can be set in HTML with an attribute named on<event>.

For instance, to assign a click handler for an input, we can use onclick, like here:

<input value="Click me" onclick="alert('Click!')" type="button">

On mouse click, the code inside onclick runs.

An HTML-attribute is not a convenient place to write a lot of code, so we’d better create a JavaScript function and call it there. Here a click runs the function countRabbits():

<script>

function countRabbits() {

for(let i=1; i<=3; i++) {

alert("Rabbit number " + i);

}

}

</script>

<input type="button" onclick="countRabbits()" value="Count rabbits!">

**Accessing the element: this**

The value of this inside a handler is the element. The one which has the handler on it.

In the code below button shows its contents using this.innerHTML:

<button onclick="alert(this.innerHTML)">Click me</button>

### Set a handler by DOM property: on<event>

We can assign a handler using a DOM property on<event>. For instance, elem.onclick:

<input id="elem" type="button" value="Click me">

<script>

elem.onclick = function() {

alert('Thank you');

};

</script>

If the handler is assigned using an HTML-attribute then the browser reads it, creates a new function from the attribute content and writes it to the DOM. So this way is the same as the previous one.

Either way, **as there’s only one onclick property, we can’t assign more than one event handler.** Adding a new handler with JavaScript overwrites the existing handler.

To remove a handler – assign elem.onclick = null.

### Set a handler by DOM method: addEventListener

Multiple handlers cannot be set for an event by on<event> property (HTML-attribute or DOM-attribute). Setting handlers for an event by addEventListener  method solves this problem.

The syntax:

element.addEventListener(event, handler, [options]);

**event** Event name, e.g. "click".

**handler** The handler function.

**Options**

* once: if true, then the listener is automatically removed after it triggers.
* capture: false|true whether the event should be executed in the capturing (true) or in the bubbling phase (false).
* passive: if true, then the handler will not call preventDefault(),

To remove the handler, use removeEventListener:

element.removeEventListener(event, handler, [options]);

Multiple calls to addEventListener allow to add multiple handlers, like this:

<input id="elem" type="button" value="Click me"/>

<script>

function handler1() {

alert('Thanks!');

};

function handler2() {

alert('Thanks again!');

}

elem.onclick = () => alert("Hello");

elem.addEventListener("click", handler1); // Thanks!

elem.addEventListener("click", handler2); // Thanks again!

</script>

For some event, like DOMContentLoaded event, handlers work with only addEventListener.

// will never run

document.onDOMContentLoaded = function() {

alert("DOM built");

};

// this way it works

document.addEventListener("DOMContentLoaded", function() {

alert("DOM built");

});

We can assign not just a function, but an object as an event handler using addEventListener. When an event occurs, its handleEvent method is called.

For instance:

<button id="elem">Click me</button>

<script>

let obj = {

handleEvent(event) {

alert(event.type + " at " + event.currentTarget);

}

};

elem.addEventListener('click', obj);

</script>

As we can see, when addEventListener receives an object as the handler, it calls obj.handleEvent(event) in case of an event.

### Event object

To properly handle an event we’d want to know more about what’s happened. Not just a “click” or a “keydown”, but what were the pointer coordinates? Which key was pressed? And so on.

When an event happens, the browser creates an event object, puts details into it and passes it as an argument to the handler.

Here’s an example of getting pointer coordinates from the event object:

<input type="button" value="Click me" id="elem">

<script>

elem.onclick = function(event) {

// show event type, element and coordinates of the click

alert(event.type + " at " + event.currentTarget);

alert("Coordinates: " + event.clientX + ":" + event.clientY);

};

</script>

Some properties of event object:

**event.type** Event type, here it’s "click".

**event.currentTarget** Element that handled the event. That’s exactly the same as this, unless the handler is an arrow function, or its this is bound to something else, then we can get the element from event.currentTarget.

**event.clientX / event.clientY** Window-relative coordinates of the cursor, for pointer events.

# Json

JSON (@ <http://json.org/>) is a lightweight, text-based, human-readable format for data-interchange. JSON is a data format, just like XML, but smaller and lighter in size than XML and, hence, is an alternative to XML. JSON format is based on JavaScript Object (and Array) syntaxes, hence, called JavaScript Object Notation.

JSON supports these data types:

* Number, String and Boolean (true or false).
* Array: an ordered, comma-separated sequence of values enclosed in square bracket [].
* Object: an unordered, comma-separated key:value pairs enclosed in curly bracket {}.
* null: for unallocated object.

For example, below is a properly JSON-formatted text:

{ "members": [

{"name":"Paul", "age":50, "isMarried":true},

{"name":"John", "age":40, "isMarried":false},

{"name":"Mary", "age":30, "isMarried":true}

]}

The data contains one key:value pair, with key of members and value of an array of three objects. Each object has three properties: name (string), age (number) and isMarried (boolean).

Notes:

* JSON's key fields must be double-quoted as shown; while in JavaScript, the quotes are optional if the key is a valid identifier.
* JSON's file type is ".json".
* JSON's MIME type is application/json.

JavaScript provides methods:

#### JSON.parse(): convert JSON into an object.

var jsonStr = '{ "members": ['

+ '{"name":"Paul", "age":50, "isMarried":true},'

+ '{"name":"Anna", "age":40, "isMarried":false},'

+ '{"name":"Peter", "age":30, "isMarried":true}'

+ ']}';

var obj = JSON.parse(jsonStr);

console.log(obj); // Object {members: Array[3]}

console.log(obj.members[0].name); // Paul

#### JSON.stringify(): convert objects into JSON.

var obj = {

students: [

{name:"Paul", "age":50, "isMarried":true},

{name:"Anna", "age":40, "isMarried":false},

{name:"Peter", "age":30, "isMarried":true}

]};

var jsonStr = JSON.stringify(obj);

console.log(jsonStr);

// {"students":[

// {"name":"Paul","age":50,"isMarried":true},

// {"name":"Anna","age":40,"isMarried":false},

// {"name":"Peter","age":30,"isMarried":true}

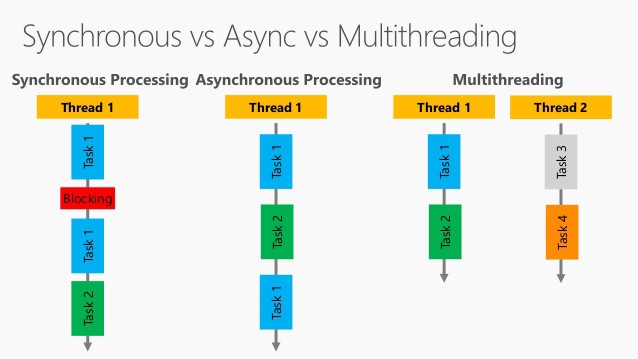
// ]}

# Ajax

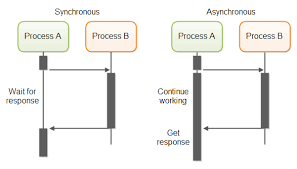
# Asynchronous programming

## Synchronous vs Asynchronous vs Multi-threading

When all tasks run on a same computer



When tasks A, B run on 2 computers but task A in this computer has to wait for task B on the other computer



## Why is Javascript asynchronous while single-threaded?

The answer is:

* Javascript is single-threaded but the browser is multi-threaded, the Javascript code is a thread and the IO/event code runs on a different thread which may be on another computer
* The browser has “event loop” mechanism to switch back and forth the Javascript code and the event code

<https://dev.to/steelvoltage/if-javascript-is-single-threaded-how-is-it-asynchronous-56gd>

* how do we get asynchronous code with Javascript then?
* Well, we can thank the Javascript engine (V8, Spidermonkey, JavaScriptCore, etc...) for that, which has Web API that handle these tasks in the background. The call stack recognizes functions of the Web API and hands them off to be handled by the browser. Once those tasks are finished by the browser, they return and are pushed onto the stack as a callback.
* Open your console and type window then press enter. You'll see most everything the Web API has to offer. This includes things like ajax calls, event listeners, the fetch API, and setTimeout. Javascript uses low level programming languages like C++ to perform these behind the scenes.
* Let's look at a simple example, run this code your console:

console.log("first")

setTimeout(() => {

console.log("second")

}, 1000)

console.log("third")

* What did we get back?

first

third

undefined

second

* Feels odd, right? Well, let's break this down line by line:
* console.log("first") is on the stack first, so it gets printed. Next, the engine notices setTimeout, which isn't handled by Javascript and pushes it off to the WebAPI to be done asynchronously. The call stack moves on without caring about the code handed off to the Web APIs and console.log("three") is printed.
* Next, the Javascript engine's event loop kicks in, like a little kid asking "Are we there yet?" on a road trip. It starts firing, waiting for events to be pushed into it. Since the setTimeout isn't finished, it returns undefined, as the default, well because it hasn't been given the value yet. Once the callback finally does hits we get console.log("second") printed.

In an asychronous environment, a single process thread runs all the time, but it may, for event-driven reasons (and that is the key), switch from one function to another. When an event happens, *and when the currently running process hits a point at which it must wait for another event*, the javascript core then scans its list of events and delivers the next one, in a (formally) indeterminate (but probably deterministic) order, to the event manager.

For this reason, event-driven, asynchronous programming avoids many of the pitfalls of traditional, multi-threaded programming, such as memory contention issues.

*Everything runs on a different thread*except*our code*

<https://www.sohamkamani.com/blog/2016/03/14/wrapping-your-head-around-async-programming/>

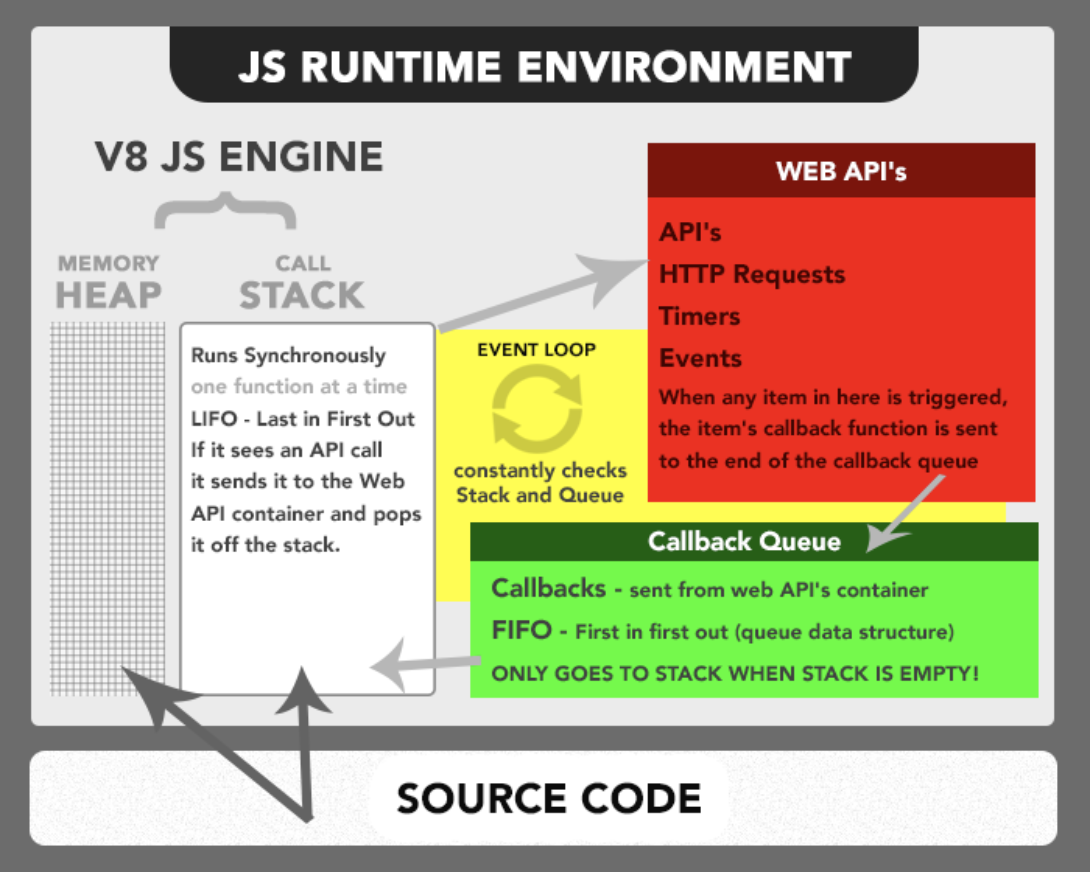
## **What is Javascript engine and Runtime?**

**Javascript runtime** refers to where your **javascript** code is executed when you run it. That said, **javascript** can be executed on google chrome, in which case your **javascript engine**is v8, if on mozilla — it is spidermonkey, if IE — then its chakra, if Safari — it’s nitro and if on node, again its v8. Now what is JS engine and what is JS runtime?

Engine converts the javascript we write into machine code. All JavaScript engines implement [**specification of the language** provide by ECMAScript](https://www.ecma-international.org/publications/standards/Ecma-262.htm). Standardisation facilitates the development of independent engines and ensures your scripts give the same results no matter where you run them. In order to obtain speed, V8 translates JavaScript code into more efficient machine code instead of using an interpreter. It compiles JavaScript code into machine code at execution by implementing a **JIT (Just-In-Time) compiler** like a lot of modern JavaScript engines do such as SpiderMonkey or Rhino (Mozilla). The main difference here is that V8 doesn’t produce bytecode or any intermediate code. JavaScript engine is just a building block of a bigger concept. This engine works inside an environment called Javascript Runtime which provides additional features to our scripts. These features can be making a call to web, catching mouse/keyboard events, etc.

This is the architecture of a JS Runtime. V8 does not have these WebAPI’s. These are given by runtime. In chrome browser JS runtime, browser has it while in Node it’s given by C++ libraries.

Image for post



Runtime Architecture

Let’s go with how is Javascript asynchronous and single threaded.

## **JS is Single Threaded but JS runtime is not. What about event loop??**

**Javascript code is executed in a single thread but Javascript runtime is not run in single thread.** **Thread pool exists in JS runtime but we don’t have to worry about it as Runtime takes care of it. But how does it does that? Event loop to the rescue.**

Lets understand what is Heap and Call stack in the Runtime (or in JS enginee which belongs to Runtime). The javascript code is first converted to machine code. Heap stores all the variables and call stack performs the operations.

console.log("Start")function sayHello(name) {  
 console.log(`Hello ${name}!`)  
}sayHello("Abhinav");console.log("End")

All these go to call stack and are executed there.

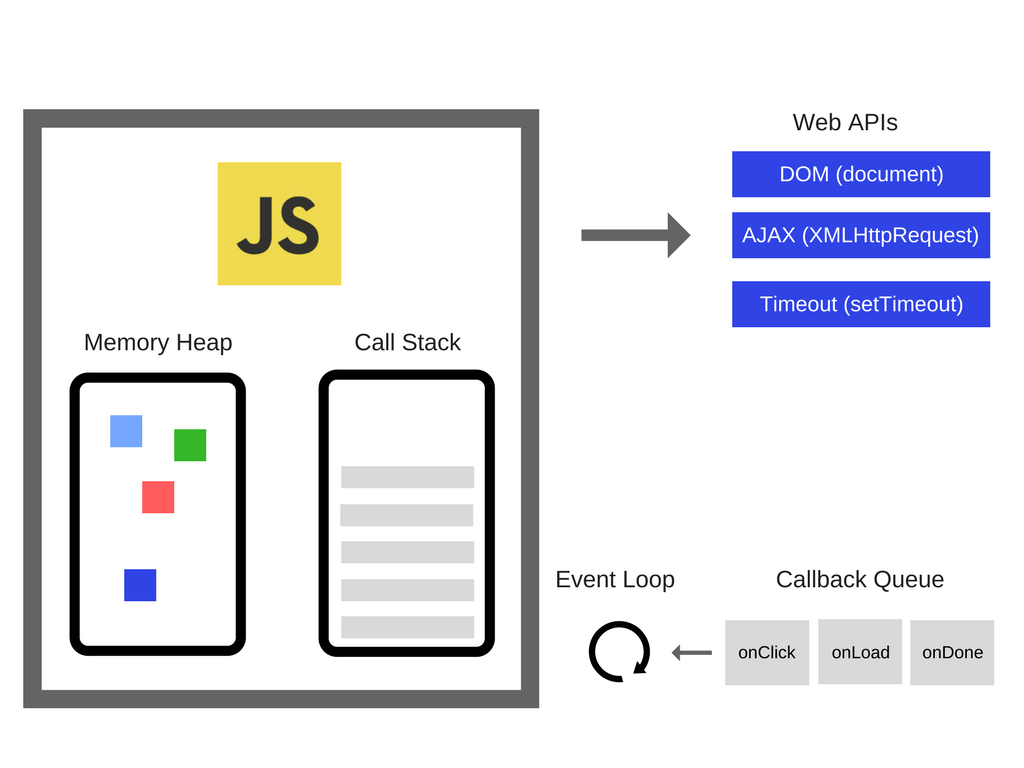
Start  
Hello Abhinav  
End

We can divide the scripts in 2 types namely — immediately invoked and invoked for later.

What happens when asynchronous tasks come? Tasks which take time to run. Say making an API call or Timer, etc. There is a concept called callback. It is the function to be executed when this task is done.

Well they go into the call stack as any normal functions but we make a call to WebAPIs as this task resided in WebAPIs. It stores the callback function for the task and does the task for us (using threading / multi processing depending on the runtime). And when the task is finished, it sends the callback to callback queue.

Visualise again here.



Now what is event loop? Event loop runs continuously (in Browser runtime which it doesn’t always runs in node) to check if the call stack is empty and if it is empty, it picks up the first item from the callback queue and moves it to call stack and executes the callback function. Until the stack is not empty, no function is added from callback queue.

Callbacks are **A L W A Y S**executed completely. **The Event loop runs one callback at a time**. No context switching. All callbacks in the queue have to wait until the current one is finished. If a script runs too long, it blocks others. That’s why **callbacks should be relatively short and simple**.

Pretty simple right! But in reality it’s much more complex. There are multiple queues depending on the runtime and their priorities are different. There is something as rendering queue. Whose job is to render the screen.

## More about Event loop

<https://www.youtube.com/watch?v=8aGhZQkoFbQ>

For instance, the  that t

Some event

Array vs Object

in web browsers or on V8 engine

Data types:

* String
* Number
* Boolean
* Object
* Function

Array vs

# MISC

<https://javascript.info/dom-navigation>

<https://cs.lmu.edu/~ray/notes/javascript/>

<https://developer.mozilla.org/en-US/docs/Web/JavaScript>