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 a compiler.

# Javascript language

# Run Javascript

## Run Javascript in browser or Node

Browser:

* Chrome, Opera: V8
* Firefox: SpiderMonkey
* IE: Chakra

## Where to put Javascript

**Javascript is placed inside <script>…</script>**

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 <script> elements in an HTML document.

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

**External javascript file**

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’s interaction with users: modal, console log(), innerHTML/write()

In HTML, one has <form> for user input and the whole HTLM page is for output; but, that is HTLM, not Javascript. This section presents only Javascript (not HTML) interaction with users.

The environment that users interact with Javascript is the browser, which has 3 components: modal, HTML, console log; users interact with those 3 components. The modal prompt(), confirm(“question”) allow users to give their input while modal alert(), console.log(), innerHTML/write() provides only output.

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

Javascript’s functions: alert, prompt and confirm will show users a modal, which is a message box in the browser that may include a textbox for user input. Seeing this modal, users cannot interact with the rest of the page; users have to enter some data/press some button on the modal to continue.

### [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");

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

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

The function prompt accepts two arguments:

result = prompt(title, [default]);

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

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

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!

### [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 cannot access output device except printer

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>

## 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>

## HTML: innerHTML or document.write()

### document.getElementById(…).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.

### 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.

# JavaScript’s syntaxes that are different from Java’s

## A lot of Javascript’s syntax is similar to Java’s

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.

Single line comments start with // and Multi-line comments start with /\* and end with \*/.

JavaScript Identifiers (names) are used to name variables, functions: the first character must be a letter (must not be a digit), or an underscore (\_), or a dollar sign ($). JavaScript is case-sensitive and uses lower camel case.

Branches and loops in Javascript are identical to those of Java.

## Javascript’s syntax that is different from Java’s

### Literal

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

Example: in the following, “A decimal number is: “ and 4.25e+6 are string and number literals

console.log("A decimal number is: " + 4.25e+6);

### Variables: var, let, const

Variables: JavaScript uses the **var, let, const** keyword to declare variables.

The scope of **let** and **const** is within the code block that contains let, const

If **var** is not within any function, its scope is global; otherwise its scope is within the function that contains it regardless of what code block the **var** is in.

Example: var globalScopeNum is global and var functionalScopeNum is within function func.

const blockScopeScale and let blockScopeIndex are local to the block containing them

<html>

<body>

    <script>

        var globalScopeNum = 1;

        function func(){

            for(let blockScopeIndex = 0; blockScopeIndex < 10; blockScopeIndex++){

                if((blockScopeIndex % 2) == 1){

                    const blockScopeScale = 3;

                    var functionalScopeNum = blockScopeIndex \* blockScopeScale;

                }

                // console.log(blockScopeScale);

//Error since blockScopeScale is within only the IF block

            }

            // console.log(blockScopeIndex);

// Error since blockScopeIndex is within only the FOR loop

            console.log(functionalScopeNum);

        }

        func();

        console.log(globalScopeNum);

    </script>

</body>

</html>

### Control structure

#### JS’s conditional (if, switch), loop (for, while, do while), jump (break, continue) are identical to Java

#### for … of, for … in

for (<variable> of <iterable values of an object>) {

// statements to be executed

}

for (<variable> of <*enumerable property keys of an object*>) {

// statements to be executed

}

Both for..in and for..of are looping constructs which are used to iterate over data structures. The only difference between them is the entities they iterate over:

1. **for..in** iterates over all enumerable **property keys of an object**
2. **for..of** iterates over the **values of an iterable object**. Examples of iterable objects are arrays, strings, and NodeLists.

Example:

let arr = ['el1', 'el2', 'el3'];

arr.addedProp = 'arrProp';

// elKey are the property keys

for (let elKey in arr) {

console.log(elKey); // 0, 1, 2, arrProp

}

// elValue are the property values

for (let elValue of arr) {

console.log(elValue) // el1, el2, el3

}

# Javascript data types

There are six basic data types in JavaScript which can be divided into three main categories:

- primitive (or *primary*): String, Number, Boolean

- *composite* (or *reference*): Object, Array, Function

- *special* data types: undefined, null

Primitive data types can hold only one value at a time, whereas composite data types can hold collections of values and more complex entities.

Primitive are immutable while composite (object, array, function) are mutable.

## Javascript data types

### Primitive: String, Number, Boolean

#### String

The string data type is used to represent textual data (i.e. sequences of characters). Strings are created using single or double quotes surrounding one or more characters, as shown below:

Example

var a = 'Hi there!'; // using single quotes

var b = "Hi there!"; // using double quotes

You can include quotes inside the string as long as they don't match the enclosing quotes.

Example

var a = "Let's have a cup of coffee."; // single quote inside double quotes

var b = 'He said "Hello" and left.'; // double quotes inside single quotes

var c = 'We\'ll never give up.'; // escaping single quote with backslash

#### Number

##### Regular number

The number data type is used to represent positive or negative numbers with or without decimal place, or numbers written using exponential notation e.g. 1.5e-4 (equivalent to 1.5x10-4).

Example

var a = 25; // integer

var b = 80.5; // floating-point number

var c = 4.25e+6; // exponential notation, same as 4.25e6 or 4250000

var d = 4.25e-6; // exponential notation, same as 0.00000425

##### Infinity, -Infinity, NaN

The Number data type also includes some special values which are: Infinity, -Infinity and NaN. Infinity represents the mathematical Infinity ∞, which is greater than any number. Infinity is the result of dividing a nonzero number by 0, as demonstrated below:

Example

alert(16 / 0); // Output: Infinity

alert(-16 / 0); // Output: -Infinity

alert(16 / -0); // Output: -Infinity

While NaN represents a special Not-a-Number value. It is a result of an invalid or an undefined mathematical operation, like taking the square root of -1 or dividing 0 by 0, etc.

Example

alert("Some text" / 2); // Output: NaN

alert("Some text" / 2 + 10); // Output: NaN

alert(Math.sqrt(-1)); // Output: NaN

#### Boolean

The Boolean data type can hold only two values: true or false. It is typically used to store values like yes (true) or no (false), on (true) or off (false), etc. as demonstrated below:

Example

var isReading = true; // yes, I'm reading

var isSleeping = false; // no, I'm not sleeping

Boolean values also come as a result of comparisons in a program. The following example compares two variables and shows the result in an alert dialog box:

Example

var a = 2, b = 5, c = 10;

alert(b > a) // Output: true

alert(b > c) // Output: false

### Special data type: undefined, null

**undefined** means a variable has been declared but has not yet been assigned a value, such as:

var testVar;

alert(testVar); //shows undefined

alert(typeof testVar); //shows undefined

**null** is an assignment value. It can be assigned to a variable as a representation of no value:

var testVar = null;

alert(testVar); //shows null

alert(typeof testVar); //shows object

From the preceding examples, it is clear that undefined and null are two distinct types: undefined is a type itself (undefined) while null is an object.

null === undefined // false

null == undefined // true

null === null // true

and

null = 'value' // ReferenceError

undefined = 'value' // 'value'

### Composite (reference): Object, Array, Function

#### Object

The object is a complex data type that allows you to store collections of data.

An object contains properties, defined as a key-value pair. A property key (name) is always a string, but the value can be any data type, like strings, numbers, booleans, or complex data types like arrays, function and other objects.

Example

var emptyObject = {};

var person = {"name": "Clark", "surname": "Kent", "age": "36"};

// You can omit the quotes around property name if the name is a valid one.

var car = {

"modal": "BMW X3",

"color": "white",

"doors": 5

}

#### Array

An array is a type of object used for storing multiple values in single variable. Each value (also called an element) in an array has a numeric position, known as its index, and it may contain data of any data type-numbers, strings, booleans, functions, objects, and even other arrays. The array index starts from 0, so that the first array element is arr[0] not arr[1].

Example

var colors = ["Red", "Yellow", "Green", "Orange"];

var cities = ["London", "Paris", "New York"];

alert(colors[0]); // Output: Red

alert(cities[2]); // Output: New York

#### Function

The function is a callable object that executes a block of code. Since functions are objects, so it is possible to assign them to variables, as shown in the example below:

Example

var greeting = function(){

return "Hello World!";

}

// Check the type of greeting variable

alert(typeof greeting) // Output: function

alert(greeting()); // Output: Hello World!

In fact, functions can be used at any place any other value can be used. Functions can be stored in variables, objects, and arrays. Functions can be passed as arguments to other functions, and functions can be returned from functions. Consider the following function:

Example

function createGreeting(name){

return "Hello, " + name;

}

function displayGreeting(greetingFunction, userName){

return greetingFunction(userName);

}

var result = displayGreeting(createGreeting, "Peter");

alert(result); // Output: Hello, Peter

## Primitve are immutable, Composite (Objects, Arrays, Functions) 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.

## typeof returns the type of a variable

The typeof operator can be used to find out what type of data a variable or operand contains. It can be used with or without parentheses (typeof(x) or typeof x).

The typeof operator is particularly useful in the situations when you need to process the values of different types differently, but you need to be very careful, because it may produce unexpected result in some cases, as demonstrated in the following example:

Example

// Numbers

typeof 15; // Returns: "number"

typeof 42.7; // Returns: "number"

typeof 2.5e-4; // Returns: "number"

typeof Infinity; // Returns: "number"

typeof NaN; // Returns: "number". Despite being "Not-A-Number"

// Strings

typeof ''; // Returns: "string"

typeof 'hello'; // Returns: "string"

typeof '12'; // Returns: "string". Number within quotes is typeof string

// Booleans

typeof true; // Returns: "boolean"

typeof false; // Returns: "boolean"

// Undefined

typeof undefined; // Returns: "undefined"

typeof undeclaredVariable; // Returns: "undefined"

// Null

typeof Null; // Returns: "object"

// Objects

typeof {name: "John", age: 18}; // Returns: "object"

// Arrays

typeof [1, 2, 4]; // Returns: "object"

// Functions

typeof function(){}; // Returns: "function"

Note: as a convention of Javascript,  typeof null is "object" instead of "null".

## Type conversion

Most of the time, operators and functions automatically convert the values given to them to the right type. For example, alert automatically converts any value to a string to show it. Mathematical operations convert values to numbers.

### [String Conversion](https://javascript.info/type-conversions" \l "string-conversion)

String conversion happens when we need the string form of a value. For example, alert(value) does it to show the value: a false becomes "false", null becomes "null", etc.

We can also call the String(value) function to convert a value to a string:

let value = true;

alert(typeof value); // boolean

value = String(value); // now value is a string "true"

alert(typeof value); // string

### [Numeric Conversion](https://javascript.info/type-conversions" \l "numeric-conversion)

Numeric conversion happens in mathematical functions and expressions automatically.

For example, when division / is applied to non-numbers:

alert( "6" / "2" ); // 3, strings are converted to numbers

We can use the Number(value) function to explicitly convert a value to a number:

let str = "123";

alert(typeof str); // string

let num = Number(str); // becomes a number 123

alert(typeof num); // number

let age = Number("an arbitrary string instead of a number");

alert(age); // NaN, conversion failed

Numeric conversion rules:

| **Value** | **Becomes…** |
| --- | --- |
| undefined | NaN |
| null | 0 |
| true and false | 1 and 0 |
| string | Whitespaces from the start and end are removed. If the remaining string is empty, the result is 0. Otherwise, the number is “read” from the string. An error gives NaN. |

Examples:

alert( Number(" 123 ") ); // 123

alert( Number("123z") ); // NaN (error reading a number at "z")

alert( Number(true) ); // 1

alert( Number(false) ); // 0

Please note that null becomes zero while undefined becomes NaN.

### [Boolean Conversion](https://javascript.info/type-conversions" \l "boolean-conversion)

The conversion rule:

* Values that are intuitively “empty”, like 0, an empty string, null, undefined, and NaN, become false.
* Other values become true.

You can call Boolean(value) to perform explicitly casting:

For instance:

alert( Boolean(1) ); // true

alert( Boolean(0) ); // false

alert( Boolean("hello") ); // true

alert( Boolean("") ); // false

**Please note: the string with zero "0" is true**

Some languages (namely PHP) treat "0" as false. But in JavaScript, a non-empty string is always true.

alert( Boolean("0") ); // true

alert( Boolean(" ") ); // spaces, also true (any non-empty string is true)

## 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

## Operators

In the following some variables start with $ like $x, $y and this is normal. Note that a variable name can start with a alphabet letter, dollar signe $ or underscore \_.

**JavaScript Arithmetic Operators**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Result** |
| + | Addition | $x + $y | Sum of $x and $y |
| - | Subtraction | $x - $y | Difference of $x and $y. |
| \* | Multiplication | $x \* $y | Product of $x and $y. |
| / | Division | $x / $y | Quotient of $x and $y |
| % | Modulus | $x % $y | Remainder of $x divided by $y |

**JavaScript Assignment Operators**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Is The Same As** |
| = | Assign | x = y | x = y |
| += | Add and assign | x += $ | x = x + y |
| -= | Subtract and assign | x -= y | x = x - y |
| \*= | Multiply and assign | x \*= y | x = x \* y |
| /= | Divide and assign quotient | x /= y | x = x / y |
| %= | Divide and assign modulus | x %= y | x = x % y |

**JavaScript String Operators**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Result** |
| + | Concatenation | str1 + str2 | Concatenation of str1 and str2 |
| += | Concatenation assignment | str1 += str2 | Appends the str2 to the str1 |

**JavaScript Incrementing and Decrementing Operators**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Name** | **Effect** |
| ++x | Pre-increment | Increments x by one, then returns x |
| x++ | Post-increment | Returns x, then increments x by one |
| --x | Pre-decrement | Decrements x by one, then returns x |
| x-- | Post-decrement | Returns x, then decrements x by one |

**JavaScript Logical Operators**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Name** | **Example** | **Result** |
| && | And | x && y | True if both x and y are true |
| || | Or | x || y | True if either x or y is true |
| ! | Not | !x | True if x is not true |

**Javascript ? operator**

let result = condition ? value1 : value2;

The condition is evaluated: if it’s truthy then value1 is returned, otherwise – value2.

For example:

let accessAllowed = (age > 18) ? true : false;

let message = (age < 3) ? 'Hi, baby!' :

(age < 18) ? 'Hello!' :

(age < 100) ? 'Greetings!' :

'What an unusual age!';

alert( message );

**JavaScript Comparison Operators**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Name** | **Example** | **Result** |
| == | Equal | x == y | True if x is equal to y (regardless their data types) |
| === | Identical | x === y | True if x is equal to y, and they are of the same [type](https://www.tutorialrepublic.com/javascript-tutorial/javascript-data-types.php) |
| != | Not equal | x != y | True if x is not equal to y (regardless their data types) |
| !== | Not identical | x !== y | True if x is not equal to y, or they are not of the same type |
| < | Less than | x < y | True if x is less than y |
| > | Greater than | x > y | True if x is greater than y |
| >= | Greater than or equal to | x >= y | True if x is greater than or equal to y |
| <= | Less than or equal to | x <= y | True if x is less than or equal to y |

# Objects, Prototype, Class and Arrays, Maps

## 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] + '.');

}

};

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

// For example, instead of

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

// we can set:

name : {

first: 'Bob',

last: 'Smith'

},

Here we are effectively creating a **sub-namespace**. To access these items you just need to chain the extra step onto the end with another dot.

person.name.first

person.name.last

* **Instantiate an empty Object instance and then add properties**

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;

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

**To get data of an object**

// use DOT notation

person.name

person.name[0]

person.age

person.interests[1]

person.bio()

person.greeting()

// use BRACKET notation

person['age']

**To set data of an object**

person.age = 45;

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

### Add new members for an existing object

Setting members doesn't just stop at updating the values of existing properties and methods; you can also create completely new members; for example:

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.

In a browser, a default global object 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. 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).

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)

That said, using global variables is generally discouraged.

## JS prototype

### What is prototype and why need it?

Recall that you can add new properties to an object at any time as shown below.

Example: Attach property to object

function Student() {

this.name = 'John';

this.gender = 'Male';

}

var studObj1 = new Student();

studObj1.age = 15;

alert(studObj1.age); // 15

var studObj2 = new Student();

alert(studObj2.age); // undefined

As you can see in the above example, age property is attached to studObj1 instance. However, age is not added to Student() function, so studObj2 instance will not have age property because it is defined only on studObj1 instance.

What to do if we want to add new properties at later stage to a function which will be shared across all the instances? The answer is prototype.

**In JavaScript, every function or object has a property named prototype by default. For a function, its prototype property can be accessed and modified. For an object, its** prototype **property is not visible, but an object has \_\_proto\_\_ property that points to prototype object of the constructor function that created this object.**

**Function and** prototype **property:**

The prototype object is special type of enumerable object to which additional properties can be added to it which will be shared across all the instances of its constructor function.

So, use prototype property of a function in the above example in order to have age properties across all the objects as shown below:

function Student() {

this.name = 'John';

this.gender = 'M';

}

Student.prototype.age = 15;

var studObj1 = new Student();

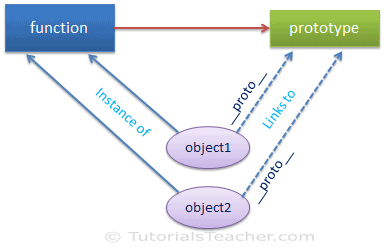
alert(studObj1.age); // 15

var studObj2 = new Student();

alert(studObj2.age); // 15

**Object and** \_\_proto\_\_**property**

Every object that is created using literal syntax or constructor syntax with the new keyword, has \_\_proto\_\_ property that points to prototype object of the constructor function that created this object.

[](https://www.tutorialsteacher.com/Content/images/oo-js/prototype-2.png)

You can debug and see object's or function's prototype property in chrome or firefox's developer tool. Consider the following example.

Example: prototype

function Student() {

this.name = 'John';

this.gender = 'M';

}

var studObj = new Student();

console.log(Student.prototype); // object

console.log(studObj.prototype); // undefined

console.log(studObj.\_\_proto\_\_); // object

console.log(typeof Student.prototype); // object

console.log(typeof studObj.\_\_proto\_\_); // object

console.log(Student.prototype === studObj.\_\_proto\_\_ ); // true

As you can see in the above example, Function's prototype property can be accessed using <function-name>.prototype. However, an object (instance) does not expose prototype property, instead you can access it using \_\_proto\_\_.

### Prototype Inheritance

#### Add properties, methods to a constructor function through prototype

In JavaScript, a prototype can be used to add properties and methods to a constructor function. And objects inherit properties and methods from a prototype. For example,

// constructor function

function Person () {

this.name = 'John',

this.age = 23

}

// creating objects

const person1 = new Person();

const person2 = new Person();

// adding property to constructor function

Person.prototype.gender = 'male';

// adding a method to the constructor function

Person.prototype.greet = function() {

console.log('hello' + ' ' + this.name);

}

// prototype value of Person

console.log(Person.prototype);

// inheriting the property from prototype

console.log(person1.gender);

console.log(person2.gender);

// inheriting the methods from prototype

person1.greet(); // hello John

person2.greet(); // hello John

Output

{ gender: "male" }

male

male

hello John

hello John

In the above program, we have added a new property gender to the Person constructor function using:

Person.prototype.gender = 'male';

Then object person1 and person2 inherits the property gender and method greet from the prototype property of Person constructor function. Hence, both objects person1 and person2 can access the gender property.

Prototype is used to provide additional property to all the objects created from a constructor function.

#### Changing Prototype

If a prototype value is changed, then all the new objects will have the changed property value. All the previously created objects will have the previous value. For example,

// constructor function

function Person() {

this.name = 'John'

}

// add a property

Person.prototype.age = 20;

// creating an object

const person1 = new Person();

console.log(person1.age); // 20

// changing the property value of prototype

Person.prototype = { age: 50 }

// creating new object

const person3 = new Person();

console.log(person3.age); // 50

console.log(person1.age); // 20

#### Inheritance hierarchy: instance – constructor’s prototype – Object’s prototype

The prototype object is being used by JavaScript engine in two things, 1) to find properties and methods of an object 2) to implement inheritance in JavaScript.

function Student() {

this.name = 'John';

this.gender = 'M';

}

Student.prototype.sayHi = function(){

alert("Hi");

};

var studObj = new Student();

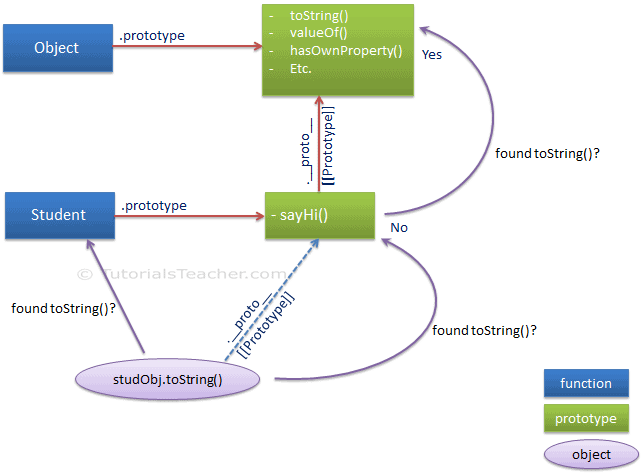
studObj.toString();

In the above example, toString() method is not defined in Student, so how and from where it finds toString()? Here, prototype comes into picture.

* First of all, JavaScript engine checks whether toString() method is attached to studObj? (It is possible to attach a new function to an instance in JavaScript).
* If it does not find there, then it uses studObj's \_\_proto\_\_ link which points to the prototype object of Student function.
* If it still cannot find it there then it goes up in the heirarchy and check prototype object of Object function because all the objects are derived from Object in JavaScript, and look for toString() method. Thus, it finds toString() method in the prototype object of Object function and so we can call studObj.toString().

This way, prototype is useful in keeping only one copy of functions for all the objects (instances).

The following figure illustrates the above scenario.

[](https://www.tutorialsteacher.com/Content/images/oo-js/prototype-3.png)

### Redirect an object’s \_\_proto\_\_

Recall: In JavaScript, objects have a special hidden property [[Prototype]] (as named in the specification), that is either null or references another object. That object is called “a prototype”:

When we read a property from object, and it’s missing, JavaScript automatically takes it from the prototype. In programming, such thing is called “prototypal inheritance”. And soon we’ll study many examples of such inheritance, as well as cooler language features built upon it.

The property [[Prototype]] is internal and hidden, but there are many ways to set it.

One of them is to use the special name \_\_proto\_\_, like this:

let animal = {

eats: true

};

let rabbit = {

jumps: true

};

rabbit.\_\_proto\_\_ = animal; // sets rabbit.[[Prototype]] = animal

Now if we read a property from rabbit, and it’s missing, JavaScript will automatically take it from animal.

For instance:

let animal = {

eats: true

};

let rabbit = {

jumps: true

};

rabbit.\_\_proto\_\_ = animal; // (\*)

// we can find both properties in rabbit now:

alert( rabbit.eats ); // true (\*\*)

alert( rabbit.jumps ); // true

Here the line (\*) sets animal to be a prototype of rabbit.

Then, when alert tries to read property rabbit.eats (\*\*), it’s not in rabbit, so JavaScript follows the [[Prototype]] reference and finds it in animal (look from the bottom up):

Here we can say that "animal is the prototype of rabbit" or "rabbit prototypically inherits from animal".

So if animal has a lot of useful properties and methods, then they become automatically available in rabbit. Such properties are called “inherited”.

If we have a method in animal, it can be called on rabbit:

let animal = {

eats: true,

walk() {

alert("Animal walk");

}

};

let rabbit = {

jumps: true,

\_\_proto\_\_: animal

};

// walk is taken from the prototype

rabbit.walk(); // Animal walk

The method is automatically taken from the prototype, like this:

The prototype chain can be longer:

let animal = {

eats: true,

walk() {

alert("Animal walk");

}

};

let rabbit = {

jumps: true,

\_\_proto\_\_: animal

};

let longEar = {

earLength: 10,

\_\_proto\_\_: rabbit

};

// walk is taken from the prototype chain

longEar.walk(); // Animal walk

alert(longEar.jumps); // true (from rabbit)

Now if we read something from longEar, and it’s missing, JavaScript will look for it in rabbit, and then in animal.

There are only two limitations:

1. The references can’t go in circles. JavaScript will throw an error if we try to assign \_\_proto\_\_ in a circle.
2. The value of \_\_proto\_\_ can be either an object or null. Other types are ignored.

Also it may be obvious, but still: there can be only one [[Prototype]]. An object may not inherit from two others.

### Object's Prototype

As mentioned before, object's prototype property is invisible. Use either \_\_proto\_\_ or Object.getPrototypeOf(obj) method to access prototype object.

Example: Object's prototype

function Student() {

this.name = 'John';

this.gender = 'M';

}

var studObj = new Student();

Student.prototype.sayHi= function(){

alert("Hi");

};

var studObj1 = new Student();

var proto = Object.getPrototypeOf(studObj1); // returns Student's prototype object

alert(proto.constructor); // returns Student function

The prototype object includes following properties and methods.

| Property | Description |
| --- | --- |
| constructor | Returns a function that created instance. |
| \_\_proto\_\_ | This is invisible property of an object. It returns prototype object of a function to which it links to. |

| Method | Description |
| --- | --- |
| hasOwnProperty() | Returns a boolean indicating whether an object contains the specified property as a direct property of that object and not inherited through the prototype chain. |
| isPrototypeOf() | Returns a boolean indication whether the specified object is in the prototype chain of the object this method is called upon. |
| propertyIsEnumerable() | Returns a boolean that indicates whether the specified property is enumerable or not. |
| toLocaleString() | Returns string in local format. |
| toString() | Returns string. |
| valueOf | Returns the primitive value of the specified object. |

## JS Class

In practice, we often need to create many objects of the same kind, like users, or goods or whatever.

As we already know from the chapter [Constructor, operator "new"](https://javascript.info/constructor-new), new function can help with that.

But in the modern JavaScript, there’s a more advanced “class” construct, that introduces great new features which are useful for object-oriented programming.

### [The “class” syntax](https://javascript.info/class" \l "the-class-syntax)

The basic syntax is:

class MyClass {

// class methods

constructor() { ... }

method1() { ... }

method2() { ... }

method3() { ... }

...

}

Then use new MyClass() to create a new object with all the listed methods.

The constructor() method is called automatically by new, so we can initialize the object there.

For example:

class User {

constructor(name) {

this.name = name;

}

sayHi() {

alert(this.name);

}

}

// Usage:

let user = new User("John");

user.sayHi();

When new User("John") is called:

1. A new object is created.
2. The constructor runs with the given argument and assigns it to this.name.

…Then we can call object methods, such as user.sayHi().

### A class is just a function

So, what exactly is a class? That’s not an entirely new language-level entity, as one might think.

Let’s unveil any magic and see what a class really is. That’ll help in understanding many complex aspects.

In JavaScript, a class is a kind of function.

Here, take a look:

class User {

constructor(name) { this.name = name; }

sayHi() { alert(this.name); }

}

// proof: User is a function

alert(typeof User); // function

What class User {...} construct really does is:

1. Creates a function named User, that becomes the result of the class declaration. The function code is taken from the constructor method (assumed empty if we don’t write such method).
2. Stores class methods, such as sayHi, in User.prototype.

After new User object is created, when we call its method, it’s taken from the prototype, just as described in the chapter [F.prototype](https://javascript.info/function-prototype). So the object has access to class methods.

We can illustrate the result of class User declaration as:

Here’s the code to introspect it:

class User {

constructor(name) { this.name = name; }

sayHi() { alert(this.name); }

}

// class is a function

alert(typeof User); // function

// ...or, more precisely, the constructor method

alert(User === User.prototype.constructor); // true

// The methods are in User.prototype, e.g:

alert(User.prototype.sayHi); // the code of the sayHi method

// there are exactly two methods in the prototype

alert(Object.getOwnPropertyNames(User.prototype)); // constructor, sayHi

### Class fields

**Old browsers may need a polyfill**

Class fields are a recent addition to the language.

Previously, our classes only had methods.

“Class fields” is a syntax that allows to add any properties.

For instance, let’s add name property to class User:

class User {

name = "John";

sayHi() {

alert(`Hello, ${this.name}!`);

}

}

new User().sayHi(); // Hello, John!

So, we just write " = " in the declaration, and that’s it.

The important difference of class fields is that they are set on individual objects, not User.prototype:

class User {

name = "John";

}

let user = new User();

alert(user.name); // John

alert(User.prototype.name); // undefined

We can also assign values using more complex expressions and function calls:

class User {

name = prompt("Name, please?", "John");

}

let user = new User();

alert(user.name); // John

### Getters/setters

Just like literal objects, classes may include getters/setters, computed properties etc.

Here’s an example for user.name implemented using get/set:

class User {

constructor(name) {

// invokes the setter

this.name = name;

}

get name() {

return this.\_name;

}

set name(value) {

if (value.length < 4) {

alert("Name is too short.");

return;

}

this.\_name = value;

}

}

let user = new User("John");

alert(user.name); // John

user = new User(""); // Name is too short.

Technically, such class declaration works by creating getters and setters in User.prototype.

### [Class Expression](https://javascript.info/class#class-expression)

Just like functions, classes can be defined inside another expression, passed around, returned, assigned, etc.

Here’s an example of a class expression:

let User = class {

sayHi() {

alert("Hello");

}

};

Similar to Named Function Expressions, class expressions may have a name.

If a class expression has a name, it’s visible inside the class only:

// "Named Class Expression"

// (no such term in the spec, but that's similar to Named Function Expression)

let User = class MyClass {

sayHi() {

alert(MyClass); // MyClass name is visible only inside the class

}

};

new User().sayHi(); // works, shows MyClass definition

alert(MyClass); // error, MyClass name isn't visible outside of the class

We can even make classes dynamically “on-demand”, like this:

function makeClass(phrase) {

// declare a class and return it

return class {

sayHi() {

alert(phrase);

}

};

}

// Create a new class

let User = makeClass("Hello");

new User().sayHi(); // Hello

### Computed names […]

Here’s an example with a computed method name using brackets [...]:

class User {

['say' + 'Hi']() {

alert("Hello");

}

}

new User().sayHi();

Such features are easy to remember, as they resemble that of literal objects.

### Making bound methods with class fields

As demonstrated in the chapter [Function binding](https://javascript.info/bind) functions in JavaScript have a dynamic this. It depends on the context of the call.

So if an object method is passed around and called in another context, this won’t be a reference to its object any more.

For instance, this code will show undefined:

class Button {

constructor(value) {

this.value = value;

}

click() {

alert(this.value);

}

}

let button = new Button("hello");

setTimeout(button.click, 1000); // undefined

The problem is called "losing this".

There are two approaches to fixing it, as discussed in the chapter [Function binding](https://javascript.info/bind):

1. Pass a wrapper-function, such as setTimeout(() => button.click(), 1000).
2. Bind the method to object, e.g. in the constructor.

Class fields provide another, quite elegant syntax:

class Button {

constructor(value) {

this.value = value;

}

click = () => {

alert(this.value);

}

}

let button = new Button("hello");

setTimeout(button.click, 1000); // hello

The class field click = () => {...} is created on a per-object basis, there’s a separate function for each Button object, with this inside it referencing that object. We can pass button.click around anywhere, and the value of this will always be correct.

That’s especially useful in browser environment, for event listeners.

### [Summary](https://javascript.info/class" \l "summary)

The basic class syntax looks like this:

class MyClass {

prop = value; // property

constructor(...) { // constructor

// ...

}

method(...) {} // method

get something(...) {} // getter method

set something(...) {} // setter method

[Symbol.iterator]() {} // method with computed name (symbol here)

// ...

}

MyClass is technically a function (the one that we provide as constructor), while methods, getters and setters are written to MyClass.prototype.

## Array

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

Javasript arrays can contain elements of different kinds.

### Create an array and access its element

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

* Literal:

var cars = ["Saab", "Volvo", 123, true];

var cars = [  
  "Saab",  
  "Volvo",  
  123,

true  
];

* 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()});

## Maps

# Functions and function objects

## Function expression and function type

Javascript’s function declaration requires “function” keyword, function names, input argument names, but no return type, no input 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;}

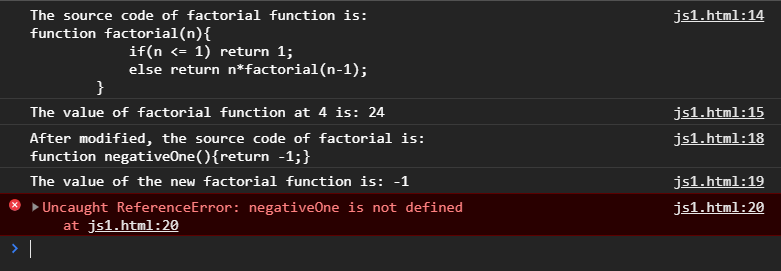
    console.log('After modified, the source code of factorial is: \n' + factorial);

    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 (without “(argument)”), 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, for example calculating factorial by recursion as below.

<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 function variable ‘x’

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

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

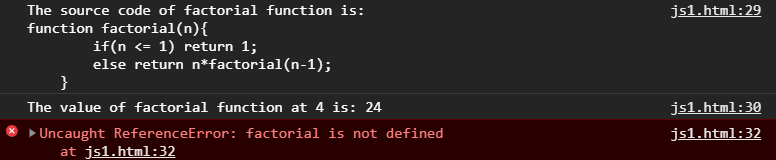
    // Show the value of factorial, factorial(4) through function name ‘factorial’ --> error

    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:



Recap:

function factorial(n){

    if(n == 1) return 1;

    else return n\*factorial(n-1);

}

// You can call a function by its function name

console.log(factorial(4));  // OK

funcVar = function anotherFactorial(n){

    if(n == 1) return 1;

    else return n\*factorial(n-1);

}

// Once you assign a function to a variable, you cannot call the function by its function name

// you have to call the function through the reference variable

console.log(anotherFactorial(4)); // error

console.log(funcVar(4)); // OK

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.

**A callback function is a function passed into another function as an *argument*, which is then invoked inside the outer function.**

Example: We’ll write a function ask(question, yesFunction, noFunction) that ask the question and, depending on the user’s answer, call yesFunction() or noFunction():

function ask(question, yesFunction, noFunction) {

if (confirm(question)) yesFunction()

else noFunction();

}

function showYes() {

alert( "You agreed." );

}

function showNo() {

alert( "You canceled the execution." );

}

// usage: functions showYes, showNo are passed as arguments to ask()

ask("Do you agree?", showYes, showNo);

The “callback” term means we pass a function and expect it to be “called back” later if necessary. In our case, showYes and showNo are callback functions.

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

function ask(question, yesFunction, noFunction) {

if (confirm(question)) yesFunction()

else noFunction();

}

ask("Do you agree?", function() { alert("You agreed."); },

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

);

**Callback functions in C/C++ and Java**

Callback function mechanism is that passing the name of function callback() to another function outer() as outer(“callback”) so that when outer() is executed, callback() is called.

**Callback in C/C++ :** Callback function mechanism is done through function pointer. Memory address of a function is represented as ‘function pointer’ in the languages like C and C++. So, the callback is achieved by passing the pointer of callback() to outer().

**Callback in Java:** In Java, we cannot pass the name of a callback function into an outer function, but we can pass an object that contains the callback function into an outer function, so the callback function mechanism can be done easily.

In the example below, we've made a callback when a button is clicked. See the steps

* Create an interface ClickEventHandler with a single method handleClick().
* Create a ClickHandler class which implements this interface ClickEventHandler.
* Create a Button class which will call ClickHandler when it's click method is called.
* Test the application.

Example

//Step 1: Create an interface for the callback handler

interface ClickEventHandler {

   public void handleClick();

}

//Step 2: Create a callback handler implementing the above interface

class ClickHandler implements ClickEventHandler {

   public void handleClick() {

      System.out.println("Clicked");

   }

}

//Step 3: Create event generator class

class Button {

   public void onClick(ClickEventHandler clickHandler) {

      clickHandler.handleClick();

   }

}

public class Tester {

   public static void main(String[] args) {

      Button button = new Button();

      ClickHandler clickHandler = new ClickHandler();

      //pass the clickHandler to do the default operation

      button.onClick(clickHandler);

      Button button1 = new Button();

      //pass the interface to implement own operation

      button1.onClick(new ClickEventHandler() {

         @Override

         public void handleClick() {

            System.out.println("Button Clicked");

         }

      });

   }

}

**Output**

Clicked Button

Clicked

## Constructor functions

Any function with the first letter capitalized can always become the constructor of an object just by being called with key word new:

<script>

function foo(){

alert('hello world');

}

let x = new foo();

</script>

However, this function above does not make a sensible constructor because it does not create any object nor assign data, etc. A better example is:

function User(name) {

this.name = name;

this.isAdmin = false;

}

let user = new User("Jack");

alert(user.name); // Jack

alert(user.isAdmin); // false

When a function is executed with new, it does the following steps:

1. A new empty object is created and assigned to this.
2. The function body executes. Usually it modifies this, adds new properties to it.
3. The value of this is returned.

In other words, new User(...) does something like:

function User(name) {

// this = {}; (implicitly)

// add properties to this

this.name = name;

this.isAdmin = false;

// return this; (implicitly)

}

So let user = new User("Jack") gives the same result as:

let user = {

name: "Jack",

isAdmin: false

};

Now if we want to create other users, we can call new User("Ann"), new User("Alice") and so on. Much shorter than using literals {…} every time, and also easy to read. That’s the main purpose of constructors – to implement reusable object creation code.

**Constructors often do not have RETURN statement**

Usually, constructors do not have a return statement. Their task is to write all necessary stuff into this, and it automatically becomes the result.

But if there is a return statement, then the rule is simple:

* If return is called with an object, then the object is returned instead of this.
* If return is called with a primitive, it’s ignored.

In other words, return with an object returns that object, in all other cases this is returned.

For instance, here return overrides this by returning an object:

function BigUser() {

this.name = "John";

return { name: "Godzilla" }; // <-- returns this object

}

alert( new BigUser().name ); // Godzilla, got that object

## “this” keyword and how to call a function

The *this* variable is attached to functions. Whenever you invoke a function, *this* is given a certain value, depending on how you invoke the function. This is often called the invocation pattern.

There are four ways to invoke functions in javascript. You can invoke the function as a *method*, as a *function*, as a *constructor*, and with *apply*.

### As a Method

A method is a function that's attached to an object

var foo = {};

foo.someMethod = function(){

alert(this);

}

When invoked as a method, *this* will be bound to the object the function/method is a part of. In this example, this will be bound to foo.

### As A Function

If you have a stand alone function, the *this* variable will be bound to the "global" object, almost always the *window* object in the context of a browser.

var foo = function(){

alert(this);

}

foo();

NOTE: In use strict mode if used as function, this is not bound to global. (It is undefined).

### As a Constructor

You can also invoke a function as a constructor with the new keyword.

function Foo(){

this.confusing = 'hell yeah';

}

var myObject = new Foo();

When invoked as a constructor, a new Object will be created, and *this* will be bound to that object, which in the above example, is myObject.

### With the Apply Method

Finally, every function has a method (yes, functions are objects in Javascript) named "apply". Apply lets you determine what the value of *this* will be, and also lets you pass in an array of arguments. Here's a useless example.

function foo(a,b){

alert(a);

alert(b);

alert(this);

}

var args = ['ah','be'];

foo.apply('omg',args);

## Anonymous functions

An anonymous function is a [function](https://www.javascripttutorial.net/javascript-function/) without a name. In the following example, the anonymous function has no name between the function keyword and parentheses ().

let show = function () {

console.log('Anonymous function');

};

show();

Because we need to call the anonymous function later, we assign the function to the show variable.

### Using anonymous functions as arguments of other functions

We often use anonymous functions as arguments of other functions. For example:

setTimeout(function () {

console.log('Execute later after 1 second')

}, 1000);

In this example, we pass an anonymous function into the [setTimeout()](https://www.javascripttutorial.net/javascript-bom/javascript-settimeout/) function. The setTimeout() function executes this anonymous function one second later.

Note that [functions are the first-class citizens](https://www.javascripttutorial.net/javascript-functions-are-first-class-citizens/) in JavaScript, so you can pass a function to another as an argument.

### Immediately invoked function execution (IIFE)

If you want to create a function and execute it immediately after declaration, you can use the anonymous function like this:

(function() {

console.log('IIFE');

})();

How it works.

First, the following defines a function expression:

(function () {

console.log('Immediately invoked function execution');

})

Second, the trailing parentheses () allow you to call the function:

(function () {

console.log('Immediately invoked function execution');

})();

and sometimes, you may want to pass arguments into it, like this:

let person = {

firstName: 'John',

lastName: 'Doe'

};

(function () {

console.log(`${person.firstName} ${person.lastName}`);

})(person);

### Arrow functions

ES6 introduced [arrow function](https://www.javascripttutorial.net/es6/javascript-arrow-function/) expression that provides a shorthand for declaring anonymous functions:

Exp1: For example, this function:

let show = function () {

console.log('Anonymous function');

};

… can be shortened using the following arrow function:

let show = () => console.log('Anonymous function');

Exp2: Similarly, the following anonymous function:

let add = function (a, b) {

return a + b;

};

… is equivalent to the following arrow function:

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

Exp3: 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

Exp 4: arrow function can be IIFE too:

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"

})(1, 2);

Summary: Anonymous functions are functions without names. Anonymous functions can be used as an argument to other functions or as an immediately invoked function execution.

## Closure

To understand closure concept, we need to look back at the scope of variable

### Lexical scope

Consider the outerFunc() and innerFunc() in the following example:

function outerFunc() {

let outerVar = 'I am outside!';

function innerFunc() {

console.log(outerVar); // => logs "I am outside!"

}

innerFunc();}

outerFunc();

Inside the innerFunc() scope, the variable outerVar is accessed from the scope of outerFunc(), which is called the lexical scope.

### Closure

Now modify the above example by moving the call to innerFunc() to outside of its lexical scop

function outerFunc() {

let outerVar = 'I am outside!';

function innerFunc() {

console.log(outerVar); // => logs "I am outside!"

}

return innerFunc;}

const myInnerFunc = outerFunc();

myInnerFunc();

Now innerFunc() is executed outside of its lexical scope. And what’s important:

*innerFunc() still has access to outerVar from its lexical scope, even being executed outside of its lexical scope.*

In other words, innerFunc() *closes over* (a.k.a. captures, remembers) the variable outerVar from its lexical scope.

In other words, innerFunc() is a *closure* because it closes over the variable outerVar from its lexical scope.

**The closure is a function that remembers the variables from the place where it is defined, regardless of where it is executed later.**

## Default, Rest parameters and Spread operator

# Javascript Modules

As our application grows bigger, we want to split it into multiple files, so called “modules”.

## A module is just a file. One script is one module

A module is just a file. One script is one module. As simple as that.

Modules can load each other and use special directives export and import to interchange functionality, call functions of one module from another one:

* export keyword labels variables and functions that should be accessible from outside the current module.
* import allows the import of functionality from other modules.

For instance, if we have a file sayHi.js exporting a function:

// 📁 sayHi.js

export function sayHi(user) {

alert(`Hello, ${user}!`);

}

…Then another file may import and use it:

// 📁 main.js

import {sayHi} from './sayHi.js';

alert(sayHi); // function...

sayHi('John'); // Hello, John!

The import directive loads the module by path ./sayHi.js relative to the current file, and assigns exported function sayHi to the corresponding variable.

Let’s run the example in-browser.

As modules support special keywords and features, we must tell the browser that a script should be treated as a module, by using the attribute <script type="module">.

Like this:

**index.html**

<!doctype html>

<script type="module">

import {sayHi} from './say.js';

document.body.innerHTML = sayHi('John');

</script>

**say.js**

export function sayHi(user) {

return `Hello, ${user}!`;

}

The browser automatically fetches and evaluates the imported module (and its imports if needed), and then runs the script.

## Core module features

What’s different in modules, compared to “regular” scripts?

There are core features, valid both for browser and server-side JavaScript.

### Always “use strict”

Modules always use strict, by default. E.g. assigning to an undeclared variable will give an error.

<script type="module">

a = 5; // error

</script>

### Module-level scope

Each module has its own top-level scope. In other words, top-level variables and functions from a module are not seen in other scripts.

In the example below, two scripts are imported, and hello.js tries to use user variable declared in user.js, and fails:

**hello.js**

alert(user); // no such variable (each module has independent variables)

**user.js**

let user = "John";

**index.html**

<!doctype html>

<script type="module" src="user.js"></script>

<script type="module" src="hello.js"></script>

Modules are expected to export what they want to be accessible from outside and import what they need.

So, we should import user.js into hello.js and get the required functionality from it instead of relying on global variables.

This is the correct variant:

**hello.js**

import {user} from './user.js';

document.body.innerHTML = user; // John

**user.js**

export let user = "John";

**index.html**

<!doctype html>

<script type="module" src="hello.js"></script>

In the browser, independent top-level scope also exists for each <script type="module">:

<script type="module">

// The variable is only visible in this module script

let user = "John";

</script>

<script type="module">

alert(user); // Error: user is not defined

</script>

If we really need to make a window-level global variable, we can explicitly assign it to window and access as window.user. But that’s an exception requiring a good reason.

### A module code is evaluated only the first time when imported

If the same module is imported into multiple other places, its code is executed only the first time, then exports are given to all importers.

That has important consequences. Let’s look at them using examples:

First, if executing a module code brings side-effects, like showing a message, then importing it multiple times will trigger it only once – the first time:

// 📁 alert.js

alert("Module is evaluated!");

// Import the same module from different files

// 📁 1.js

import `./alert.js`; // Module is evaluated!

// 📁 2.js

import `./alert.js`; // (shows nothing)

In practice, top-level module code is mostly used for initialization, creation of internal data structures, and if we want something to be reusable – export it.

Now, a more advanced example.

Let’s say, a module exports an object:

// 📁 admin.js

export let admin = {

name: "John"

};

If this module is imported from multiple files, the module is only evaluated the first time, admin object is created, and then passed to all further importers.

All importers get exactly the one and only admin object:

// 📁 1.js

import {admin} from './admin.js';

admin.name = "Pete";

// 📁 2.js

import {admin} from './admin.js';

alert(admin.name); // Pete

// Both 1.js and 2.js imported the same object

// Changes made in 1.js are visible in 2.js

So, let’s reiterate – the module is executed only once. Exports are generated, and then they are shared between importers, so if something changes the admin object, other modules will see that.

Such behavior allows us to configure modules on first import. We can setup its properties once, and then in further imports it’s ready.

For instance, the admin.js module may provide certain functionality, but expect the credentials to come into the admin object from outside:

// 📁 admin.js

export let admin = { };

export function sayHi() {

alert(`Ready to serve, ${admin.name}!`);

}

In init.js, the first script of our app, we set admin.name. Then everyone will see it, including calls made from inside admin.js itself:

// 📁 init.js

import {admin} from './admin.js';

admin.name = "Pete";

Another module can also see admin.name:

// 📁 other.js

import {admin, sayHi} from './admin.js';

alert(admin.name); // Pete

sayHi(); // Ready to serve, Pete!

### import.meta

The object import.meta contains the information about the current module.

Its content depends on the environment. In the browser, it contains the url of the script, or a current webpage url if inside HTML:

<script type="module">

alert(import.meta.url); // script url (url of the html page for an inline script)

</script>

### In a module, “this” is undefined

That’s kind of a minor feature, but for completeness we should mention it.

In a module, top-level this is undefined.

Compare it to non-module scripts, where this is a global object:

<script>

alert(this); // window

</script>

<script type="module">

alert(this); // undefined

</script>

# 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() {

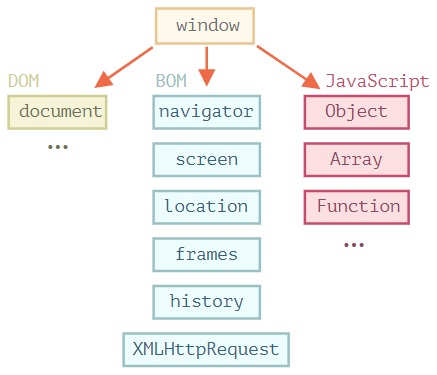
alert("Hello");

}

// 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

### DOM tree and representative JS objects

#### DOM tree with 4 kind of nodes: document, tag nodes, text nodes, comment nodes

A HTML page is structured as a tree with 4 kind of nodes:

* **Root node** is the whole HTML document; it’s represented by a Javascript object named **document**.
* Other nodes, called **element nodes**, are HTML elements like <html>, <head>, <body>, <h1>, <p>. Those elements are often nested, which makes the hierarchy structure for the tree; for example: <head>, <body> are child nodes of the root node <html>, and <h1>, <p> are child nodes of <body>.
* Leaf nodes, called **text nodes**, are the plain texts inside elements. Note that spaces and newlines are totally valid characters, like letters and digits, so they form text nodes. For example, for the node <p>Hello <b>Peter</b>, how are <i>you</i> doing?</p>, there are 5 text nodes: “Hello ”, “Peter”, “, how “, “you”, “ doing?” (don’t think it’s only one text “Hello Peter, how are you doing?”).
* **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 1: The DOM tree (starting from HTML node, instead of document) of the following HTML page

<!DOCTYPE HTML>

<html>

<head>

<title>About elk</title>

</head>

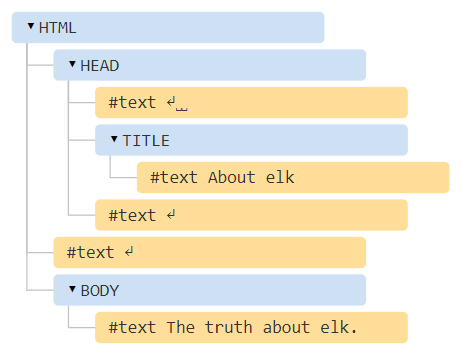
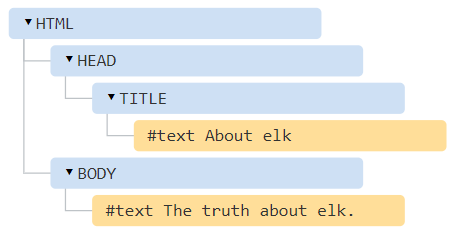
<body>

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>

Example 2: The DOM tree (starting from HTML node, instead of document) of

<html><head></head><body>The <b>truth</b> is harsh; <i>accept</i> it.</body></html>

is:

<HTML>

* <head>
* <body>
  + #text: The
  + <b>
    - #text: truth
  + #text: is harsh;
  + <i>
    - #text: accept
  + #text: it.

#### Javascript objects representing DOM nodes

The purpose of Javascript is to change elements of the HTML page, so it’s crucial for Javascript to access parts of the page.

For every HTLM page, browser creates Javascript objects to represent nodes in the DOM tree.

If an element node has the id property set, e.g. <p id=”introPara”>, <h1 id=”epilog”>, then a global variable is created with the id as its name, so by calling this global variable, one can access this HTML element. Alternatively, one can use document.getElementById(id) to get the Javascript object that represents the element node.

<html>

<body>

    <p id="firstPara">First paragraph is this</p>

    <p id="second paragraph">Second paragraph is that</p>

    <p id="third paragraph">Third paragraph is ...</p>

    <script>

        firstPara.innerHTML = "1st";

        window['second paragraph'].innerHTML = "2nd";

        document.getElementById('third paragraph').innerHTML = "3rd";

    </script>

What about DOM nodes without id?

There are 4 javascript objects that represent highest level nodes of a HTML page.

* document 🡪 the whole HTML page
* document.documentElement 🡪 the <HTML> element
* document.head 🡪 the <head>
* document.body 🡪 the <body>

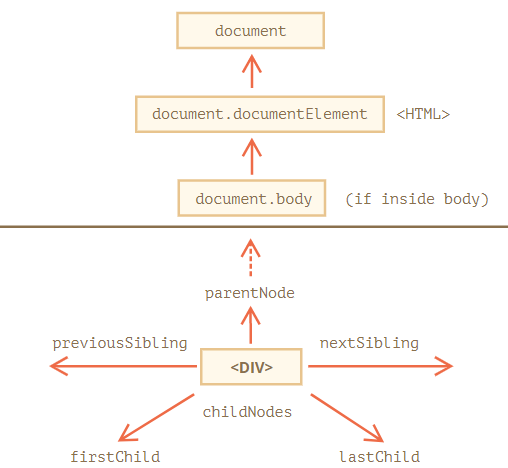
Recall that with a tree data structure, when we have the access to the root element, we can visit any node we want by searching. Here we have those 4 highest level nodes above, from them, with navigation operations (visiting child, parent, sibling) we can visit any node we want.

### Navigate to a desired DOM node

To access any node of the DOM tree, one needs to remember only:

* 4 highest nodes represented by: document, document.documentElement, document.head, document.body
* Properties to navigate to 3 levels of a node: **parent** (parentNode), **sibling** (nextSibling, previousSibling), **child** (childNodes[i], firstChild, lastChild).

As depicted in the following picture:



#### Highest level with direct access: document.documentElement and document.body

The root of the DOM tree is document; the next 3 tree nodes under document are available directly as document properties:

**<html> = document.documentElement**

**<body> = document.body**

**<head> = document.head**

To access other 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>

#### Access indirectly through navigation

##### Children: childNodes[i], firstChild, lastChild

Note: a child is an immediate descendant.

**childNodes is a collection consisting of all child nodes, including text child nodes; children  is a collection consisting of all element child nodes (i.e. no text child nodes). One can access a child node or an element child node through an index, e.g.** document.body.childNodes[i], document.body.children[i]

Note that though childNodes, children have length property and its element can be accessed through indices, childNodes, children are 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 child node.

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 (i.e. non-text 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

Siblings are nodes that are children of the same parent.

For instance, here <head> and <body> are siblings that share same parent <html>:

<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.

A sibling can be a text node; to indicate an element node (i.e. non-text node), one uses nextElementSibling previousElementSibling.

// after <head> goes <body>

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

// before <body> goes <head>

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

##### Parent

The parent of a node is available as parentNode. For example:

// parent of <body> is <html>

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

|  |
| --- |
| 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. |

#### An additional way to access a table’s elements

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 a node by element Id or by CSS selector

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?

#### Search a node by its ID: document.getElementById or just call the id directly

When an element is set up with an id attribute, a global variable whose name is the id is created and refers to the element id, so one can call this global variable to access the element.

<div id="elem">

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

</div>

// when an id is set for an element, a global variable whose name is id is created

<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>

However, this programming may cause name conflict; some people prefer method document.getElementById(id) which return the object that has the id property set.

<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>

#### Search (many) nodes by CSS selectors (querySelectorAll)

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

elem.querySelectorAll(css)[0], which returns the first element for the given CSS selector, can be shortened as elem.querySelector(css)

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.

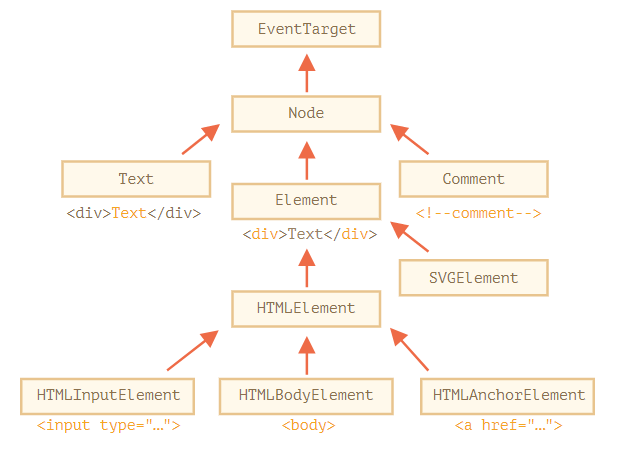
**Can use pseudo-classes as well**

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).

### Node properties: type/name and contents

How node classes inherit properties from the higher?

Each Javascript object that represent a DOM node is an instance of a corresponding built-in Javascript class. For example, a Javascript object that represents a DIV element 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

#### node content

##### element node’s content: innerHTML, outerHTML, textContent

**innerHTML**

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.

**outerHTML**

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

**textContent**

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

Example: Given this HTML fragment:

<div id="divA">This is <span>some</span> text!</div>

... you can use textContent to get the element's text content:

let text = document.getElementById('divA').textContent;

// The text variable is now: **'This is some text!**'

... or set the element's text content:

document.getElementById('divA').textContent = 'This text is different!';

// The HTML for divA is now:

// **<div id="divA">This text is different!</div>**

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

The innerHTML is only valid for element nodes. For text nodes, use: nodeValue and data properties; here we use data because it’s shorter. Note that nodeValue for an element node will return null.

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>

##### Input node’s content: value

For input nodes, you have to use “value” property to get the content of the input node.

Example

<body>

    Gender:

    <input type="radio" name="gender" value="Male">Male

    <input type="radio" name="gender" value="Female">Female

    <input type="radio" name="gender" value="Others">Others

    <br>

    <button type="button" onclick="displayRadioValue()">

        Submit

    </button>

    <br>

    <div id="result"></div>

    <script>

        function displayRadioValue() {

            var ele = document.getElementsByName('gender');

            for(i = 0; i < ele.length; i++) {

                if(ele[i].checked)

                document.getElementById("result").innerHTML

                        = "Gender: " + ele[i].value;

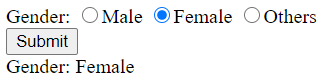
            }

        }

    </script>

</body>

Output:



## Events

### Event-driven programming: even listener listens for events and trigger a corresponding event handler callback

#### Event-driven programming in general

An **event** is a signal that something has changed; for example, user actions (mouse clicks, key presses), sensor outputs, or message passing from other programs or threads.

An **event handler** is a callback function that deals with event.

An **event listener** is a loop that listen for (detect) events and then triggers an event handler callback function when one of those events is detected.

Event-driven programming is a programming paradigm in which the flow of the program is determined by events.

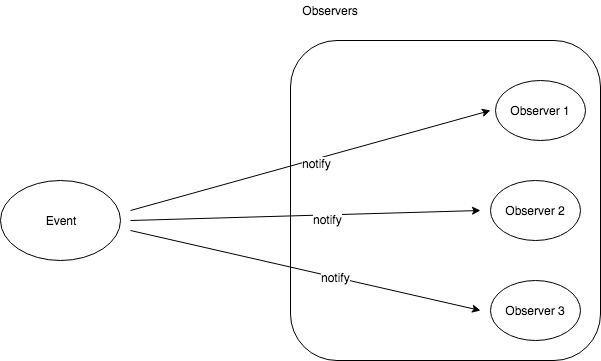
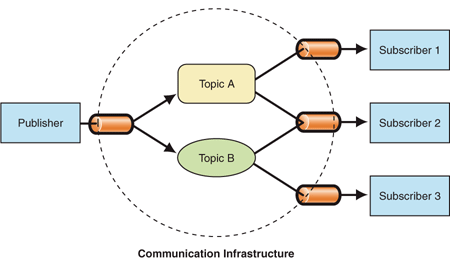
More on event-driven programming:

The most difficult part is recognizing when an event happens and what change the event makes. However, programmers don’t need to do this; Java has a mechanism to generate an event object whenever an event happens. Programmers only need to register an event listener for events and indicate what event handlers that the event listener calls.

Why need event listener while one can associate directly an event handler with the event? The answer: one more middle layer makes things more lose-coupling! Specifically, many event listeners can register for one event, one event listener can register for many events. The difference between event-eventhandler vs event-eventlistener-eventhandler is exactly the observer vs pub-sub design pattern.

**Observer pattern** is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

**Publish Subscribe** is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead categorize published messages into classes without knowledge of which subscribers, if any, there may be. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are.  
In other word the publisher and the subscriber will never know about the existence of one another. So how they communicate? There another component named message broker which is known by both publisher and subscriber. The publisher will send the message to the message broker and the message broker will filtering and broadcasting the message to the right subscriber.

The event listener here is the broker in the pub-sub design pattern.

#### Event-driven programming in Javascript

##### 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:

* transitioned – when a CSS-animation finishes.

##### Event handler and Event Listener

In Javascript, event handler can be assigned directly to events (Java does not allow this).

Also, one can use event listeners to associate multiple event handlers with multiple events, which is more flexible than the direct way of assigning one event handler – one event. In the case of event listener, event handlers are callback functions of event listener.

### Assign directly one event handler to one event by on<event> property

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

A handler can be set in HTML with an attribute named on<event> (here <event> are the names listed in the previous section).

For instance, to assign a handler for click event of 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!">

**~~(Not sure about the following~~**

**~~Accessing the element: this~~**

~~Inside a handler, you can use this to refer to the element which detected the event.~~

~~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.

### Use event listener to assign multiple event handlers to multiple events

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.

Exp 1: a JSON text that has one record:

{ "members": [

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

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

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

]}

Exp 1: a JSON text that has several records:

[

{

"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 string 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 string.

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

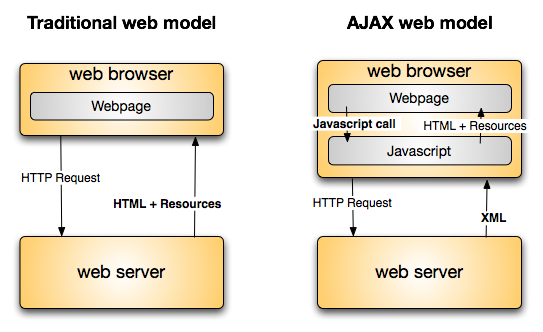
## Ajax = loading data from server without reloading ?

Ajax stands for **A**synchronous **J**avascript **A**nd **X**ml. Ajax is just a means of loading data from the server and selectively updating parts of a web page without reloading the whole page.

Basically, what Ajax does is make use of the **browser's built-in XMLHttpRequest (XHR) object** to send and receive information to and from a web server **asynchronously**, in the background. Asynchronous means execution of the script continues as soon as the Ajax request is sent, i.e. the browser will not halt the script execution until the server response comes back.

Ajax has become so popular that you hardly find an application that doesn't use Ajax to some extent. The example of some large-scale Ajax-driven online applications are: Gmail, Google Maps, Google Docs, YouTube, Facebook, Flickr, and so many other applications.

**Tip:** Don't get confused by the term **X** (i.e. **XML**) in AJAX or XMHttpRequest. It is only there for historical reasons. Other data exchange format such as JSON, HTML, or plain text can be used instead of XML.



## Sending Request and Retrieving the Response

Step 1: instantiate a request

Before you perform Ajax communication between client and server, the first thing you must do is to instantiate an XMLHttpRequest object, as shown below:

var request = new XMLHttpRequest();

Step 2: establish a connection to the server

Now, the next step in sending the request to the server is to instantiating the newly-created request object using the open() method of the XMLHttpRequest object.

The open() method typically accepts two parameters— the HTTP request method to use, such as "GET", "POST", etc., and the URL to send the request to, like this:

request.open("GET", "info.txt"); -Or- request.open("POST", "add-user.php");

**Tip:** The file can be of any kind, like .txt or .xml, or server-side scripting files, like .php or .asp, which can perform some actions on the server (e.g. inserting or reading data from database) before sending the response back to the client.

Step 3: send the request to the server

And finally send the request to the server using the send() method of the XMLHttpRequest object.

request.send(); -Or- request.send(body);

**Note:** The send() method accepts an optional body parameter which allow us to specify the request's body. This is primarily used for HTTP POST requests, since the HTTP GET request doesn't have a request body, just request headers.

The GET method is generally used to send small amount of data to the server. Whereas, the POST method is used to send large amount of data, such as form data.

In GET method, the data is sent as URL parameters. But, in POST method, the data is sent to the server as a part of the HTTP request body. Data sent through POST method will not visible in the URL.

See the chapter on [HTTP GET vs. POST](https://www.tutorialrepublic.com/php-tutorial/php-get-and-post.php) for a detailed comparison of these two methods.

In the following section we'll take a closer look at how Ajax requests actually works.

## Performing an Ajax GET Request

The GET request is typically used to get or retrieve some kind of information from the server that doesn't require any manipulation or change in database, for example, fetching search results based on a term, fetching user details based on their id or name, and so on.

The following example will show you how to make an Ajax GET request in JavaScript.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>JavaScript Ajax GET Demo</title>

<script>

function displayFullName() {

// Creating the XMLHttpRequest object

var request = new XMLHttpRequest();

// Instantiating the request object

request.open("GET", "greet.php?fname=John&lname=Clark");

// Defining event listener for readystatechange event

request.onreadystatechange = function() {

// Check if the request is compete and was successful

if(this.readyState === 4 && this.status === 200) {

// Inserting the response from server into an HTML element

document.getElementById("result").innerHTML = this.responseText;

}

};

// Sending the request to the server

request.send();

}

</script>

</head>

<body>

<div id="result">

<p>Content of the result DIV box will be replaced by the server response</p>

</div>

<button type="button" onclick="displayFullName()">Display Full Name</button>

</body>

</html>

When the request is asynchronous, the send() method returns immediately after sending the request. Therefore you must check where the response currently stands in its lifecycle before processing it using the readyState property of the XMLHttpRequest object.

The readyState is an integer that specifies the status of an HTTP request. Also, the function assigned to the onreadystatechange event handler called every time the readyState property changes. The possible values of the readyState property are summarized below.

|  |  |  |
| --- | --- | --- |
| **Value** | **State** | **Description** |
| 0 | UNSENT | An XMLHttpRequest object has been created, but the open() method hasn't been called yet (i.e. request not initialized). |
| 1 | OPENED | The open() method has been called (i.e. server connection established). |
| 2 | HEADERS\_RECEIVED | The send() method has been called (i.e. server has received the request). |
| 3 | LOADING | The server is processing the request. |
| 4 | DONE | The request has been processed and the response is ready. |

**Note:** Theoretically, the readystatechange event should be triggered every time the readyState property changes. But, most browsers do not fire this event when readyState changes to 0 or 1. However, all browsers fire this event when readyState changes to 4 .

The status property returns the numerical HTTP status code of the XMLHttpRequest's response. Some of the common HTTP status codes are listed below:

* 200 — OK. The server successfully processed the request.
* 404 — Not Found. The server can't find the requested page.
* 503 — Service Unavailable. The server is temporarily unavailable.

Please check out the [HTTP status codes](https://www.tutorialrepublic.com/html-reference/http-status-codes.php) reference for a complete list of response codes.

Here's the code from our "greet.php" file that simply creates the full name of a person by joining their first name and last name and outputs a greeting message.

<?php

if(isset($\_GET["fname"]) && isset($\_GET["lname"])) {

$fname = htmlspecialchars($\_GET["fname"]);

$lname = htmlspecialchars($\_GET["lname"]);

// Creating full name by joining first and last name

$fullname = $fname . " " . $lname;

// Displaying a welcome message

echo "Hello, $fullname! Welcome to our website.";

} else {

echo "Hi there! Welcome to our website.";

}

?>

## Performing an Ajax POST Request

The POST method is mainly used to submit a form data to the web server.

The following example will show you how to submit form data to the server using Ajax.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>JavaScript Ajax POST Demo</title>

<script>

function postComment() {

// Creating the XMLHttpRequest object

var request = new XMLHttpRequest();

// Instantiating the request object

request.open("POST", "confirmation.php");

// Defining event listener for readystatechange event

request.onreadystatechange = function() {

// Check if the request is compete and was successful

if(this.readyState === 4 && this.status === 200) {

// Inserting the response from server into an HTML element

document.getElementById("result").innerHTML = this.responseText;

}

};

// Retrieving the form data

var myForm = document.getElementById("myForm");

var formData = new FormData(myForm);

// Sending the request to the server

request.send(formData);

}

</script>

</head>

<body>

<form id="myForm">

<label>Name:</label>

<div><input type="text" name="name"></div>

<br>

<label>Comment:</label>

<div><textarea name="comment"></textarea></div>

<p><button type="button" onclick="postComment()">Post Comment</button></p>

</form>

<div id="result">

<p>Content of the result DIV box will be replaced by the server response</p>

</div>

</body>

</html>

If you are not using the FormData object to send form data, for example, if you're sending the form data to the server in the query string format, i.e. request.send(key1=value1&key2=value2) then you need to [explicitly set the request header](https://www.tutorialrepublic.com/codelab.php?topic=javascript&file=set-request-header) using setRequestHeader() method, like this:

request.setRequestHeader("Content-type", "application/x-www-form-urlencoded");

The setRequestHeader() method, must be called after calling open(), but before calling send().

Some common request headers are: application/x-www-form-urlencoded, multipart/form-data, application/json, application/xml, text/plain, text/html, and so on.

**Note:** The FormData object provides an easy way to construct a set of key/value pairs representing form fields and their values which can be sent using XMLHttpRequest.send() method. The transmitted data is in the same format that the form's submit() method would use to send the data if the form's encoding type were set to multipart/form-data.

Here's the code of our "confirmation.php" file that simply outputs the values submitted by the user.

<?php

if($\_SERVER["REQUEST\_METHOD"] == "POST") {

$name = htmlspecialchars(trim($\_POST["name"]));

$comment = htmlspecialchars(trim($\_POST["comment"]));

// Check if form fields values are empty

if(!empty($name) && !empty($comment)) {

echo "<p>Hi, <b>$name</b>. Your comment has been received successfully.<p>";

echo "<p>Here's the comment that you've entered: <b>$comment</b></p>";

} else {

echo "<p>Please fill all the fields in the form!</p>";

}

} else {

echo "<p>Something went wrong. Please try again.</p>";

}

?>

For security reasons, browsers do not allow you to make cross-domain Ajax requests. This means you can only make Ajax requests to URLs from the same domain as the original page, for example, if your application is running on the domain "mysite.com", you cannot make Ajax request to "othersite.com" or any other domain. This is commonly known as same origin policy.

However, you can load images, style sheets, JS files, and other resources from any domain.

**Tip:** Check out the [jQuery Ajax methods](https://www.tutorialrepublic.com/jquery-tutorial/jquery-ajax-get-and-post-requests.php) for quick and seamless Ajax implementation. The jQuery framework provides very convenient methods to implement Ajax functionality.

# Other Notes

Javascript is Object-based programming

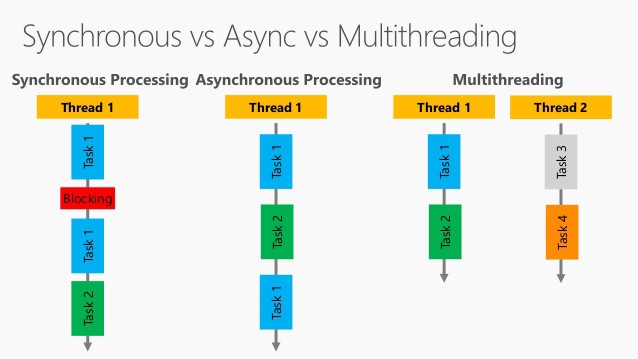
Typescript is truly OOP

JS is both loosely and dynamically typed language.

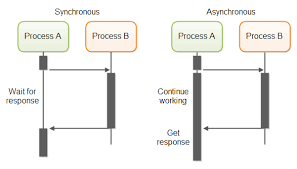
## Asynchronous programming

### 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/>

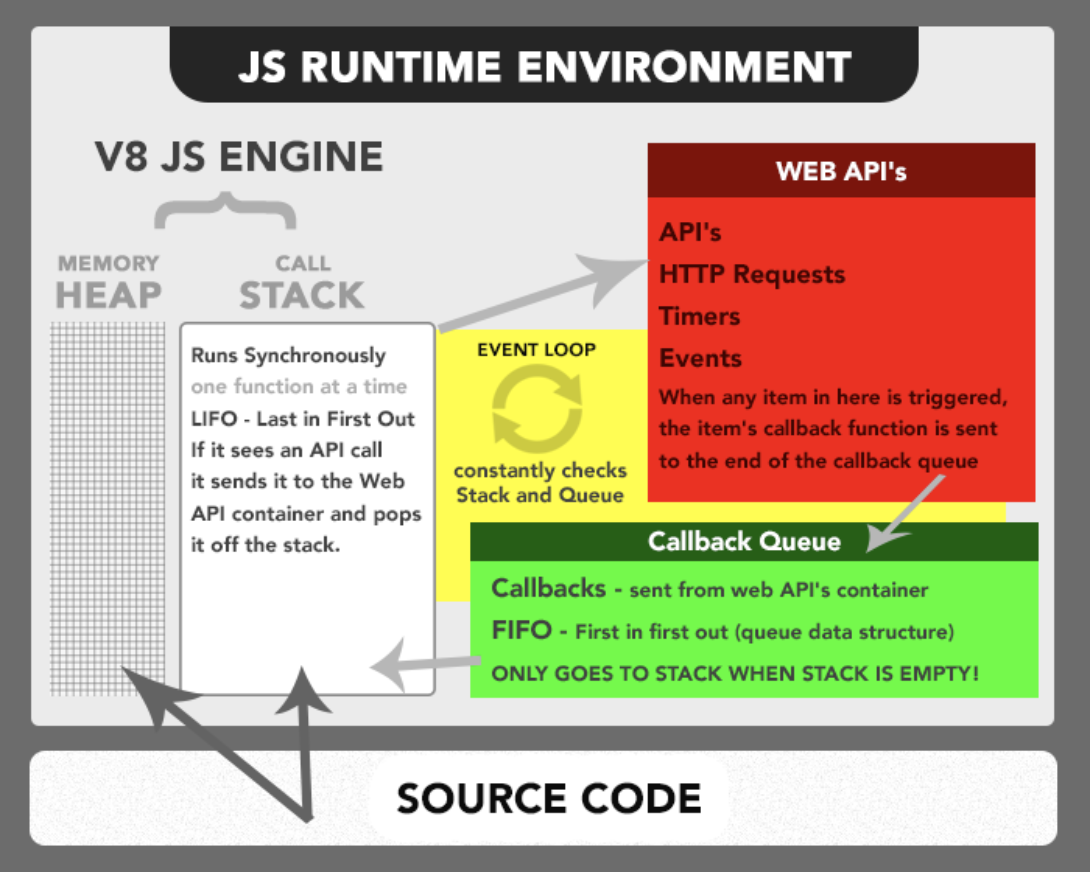
### What is Javascript engine and Runtime?

**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.

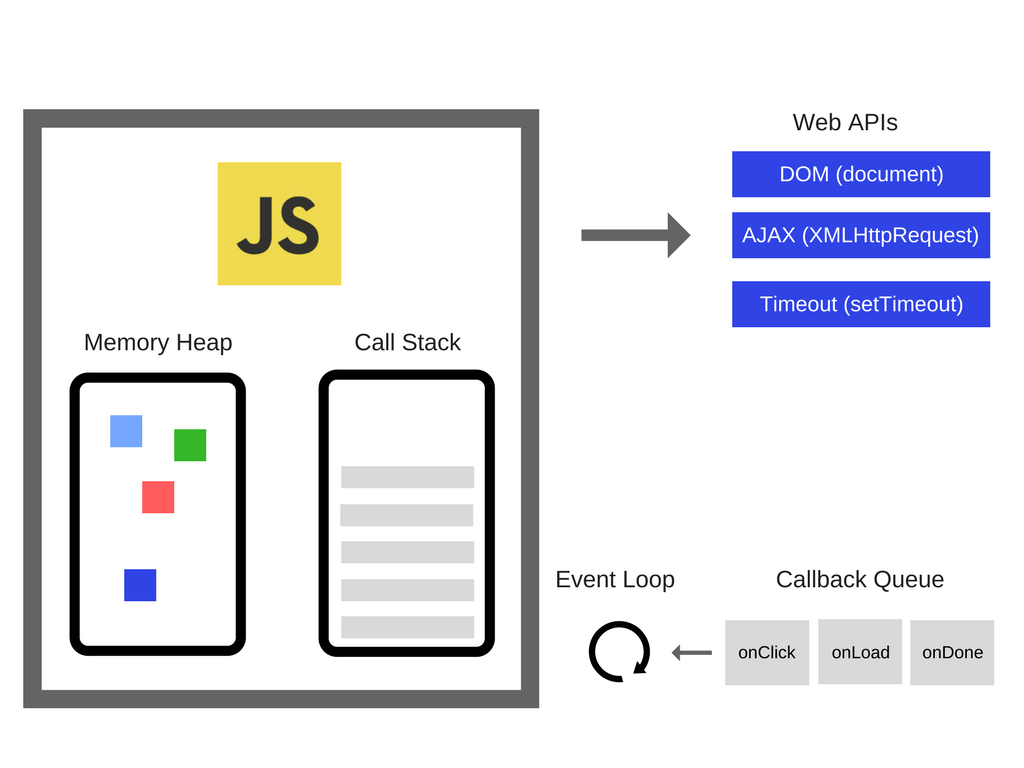
Start  
Hello Abhinav  
End

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

# Javascript OLD

## HTML DOM (Document Object Model)

### DOM allow Javascript to change content, structure, styles of HTML doc

Document Object Model represents every element in a HTML document as objects so that JavaScript can access and change the content, structures, styles of the HTML document.

Javascript can change, react, create, remove elements, attributes, styles.

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**:

### The HTML DOM Tree of Objects



### getElementById method and innerHTML property are used to change element

Example

<html>  
<body>  
  
<p id="demo"></p>  
  
<script>  
document.getElementById("demo").innerHTML = "Hello World!";  
</script>  
  
</body>  
</html>

## HTML with Javascript

Attributes for triggering something when an event occurs for any element (not only button, but also h1, img, etc): onclick, onmouseover, …

## Callback

A callback is a function passed as an argument to another function.

Exp: In the following, myDisplayer is the name of a function and it is passed to myCalculator() as an argument.

function myDisplayer(some) {  
  document.getElementById("demo").innerHTML = some;  
}  
  
function myCalculator(num1, num2, myCallback) {  
  let sum = num1 + num2;  
  myCallback(sum);  
}  
  
myCalculator(5, 5, myDisplayer);

When you pass a function as an argument, remember not to use parenthesis.

## Javascript asynchronous

# TypeScript for Javascript developers

<https://thenextweb.com/news/a-comprehensive-guide-to-typescript-for-javascript-developers>

# MISC

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

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

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