**JavaScript**

* In 2009 it became a cross-platform language, which can be used to create programs for desktops, cellphones, and the web.
* In 2009 Node.js was introduced, which is an open-source, cross-platform, back-end JavaScript runtime environment that runs on Google's V8 engine and executes JavaScript code outside a web browser.
* developer.mozilla.org/en-US/docs/Glossary/JavaScript // documentation
* jsbin.com // To test and run codes
* electron js: Electron is a free and open-source software framework developed and maintained by GitHub. It allows for the development of desktop GUI applications using web technologies: it combines the Chromium rendering engine and the Node.js runtime.
* In JavaScript, you can pass functions around as values.
* Names of classes and constructor functions are written in upper case to distinguish them from regular functions.

**ECMAScript**

* for standardization of JavaScript. So instead of having so many JavaScript versions, we can have a standard, and if a company wants to add something to JavaScript, they have to suggest it to ECMAScript, and they may confirm it.

**Validation**

* parseInt(var)
* isNaN

**Operators**

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

* + - % \* /
* = < > <= >=
* == === !
* ++x x++ --x x-- // The second one returens x firstly and then sets x
* += \*= /=
* && ||
* & |

**Variables**

* var, let in ecmascript6
* var variable = function() { return x } // **Anonymous** functions as variable values, they are like defining a function with the usual method. The name of the variable is called because it is considered to be a function.

var func = function () { return x } == function func { return x}

Func() ; // for both ways to call the function

**Scopes**

* blocks
* Global: If a global var is defined in one block, we can't access it in another one. It returns undefined if it is accessed before a local declaration in that block
* Local
* "use strict" // Write exactly this at the top of the file. It makes **var** and **let** keywords mandatory when making a variable, so we can't miss it.

**Conditions**

* If/else if/else
* switch/case/break/default. // When the number of ifs is more than 3, and they usually have the same pattern, we use switch/case. Every case value is compared by the condition and returns true or false, (x>y) == condition, for example.

switch(**condition**){

case **x** :

case y :

Do;

Break;

case z :

Do;

Break;

}

switch(condition){

case (x>y) :

Do;

Break;

default:

Do;

Break;

}

**Loops**

* For loop // It is usually used for the arrays; declaring i in the loops with var keyword makes it accessible out of the loop block and returns the last value, which is not good.
* While
* Do while: is executed once at least; for example: make a unique password and compare it with DB passwords and repeat until it is unique.

**Arrays**

* [x,y]
* Lists can have **duplicate** items
* We can add an item to an index that is bigger than the length of the array. It keeps the previous indexes empty.

var list = [1,2,3]

list[2] = true

list[6] = 'me'

[5, 2, true, **empty × 3**, 'me'] // 3 indexes don’t have any value

**list.properties**

* **list.lenght**
* **list.forEach(callback)** // takes a list and separates all the items one by one, then gives it to the Anonymous function

این آیتم ها روforeach خودش می ده به فانکشن بینام اون کاری که می خوایم رو انجام می دیم، روی آیتم ها جلو می ره و دیگه نیاز به استفاده از طول نیست.

var total = 0

array.forEach(function(item){

Total += item

});

* **list.reduce(callback(prev,item))** // Parameters are the previously calculated value and the current item of the list. The previous value at the beginning is the default value, then on the second iteration, it is the value of the first calculation, and so on. It returns the last calculated value.

var total = array.reduce(function(total, value){

**return total + value** // any calculation

}, **10**);

این آیتم ها رو reduce خودش یه پردازشی می کنه می ده به فانکشن بینام اون کاری که می خوایم رو انجام می دیم، روی آیتم ها جلو می ره و دیگه نیاز به استفاده از طول نیست. باید برای total یک مقدار اولیه بدیم وگرنه اولین آیتم رو می گیره. یک پارمتر دیگه داره که بعد فانکشن بی نام قرار می گیره که همون total هست. تو این مثال 10 .

* **list.push(item)** // adds to the right of the array and returns the last index.
* **list.pop()** // deletes the last item from the right side of the list and returns the item.
* **list.unshift(item)** // adds to the left of the list and returns the length of the list.
* **list.shift()** // deletes the last item from the left of the list and returns the item.
* **list.indexOf(item)** // returns the index of the item if it exists. -1 if the item is not in the array.
* **list.splice(startindex,howmanyitems)** // removes that part from the main list and returns it as a new list
* **list.slice(startindex,endindex)** // returns that section as a new list and the original list remains the same
* **list.join('char')** // Joins items together and returns it as a string with desired separator char. Default char is a comma.
* **list.concat(list2)** // returns a new combined list of the two [list][list2], it doesn't affect the original lists
* **list.sort()** // sorts the **main list**, it has many options
* **list.reverse()** // reverses the **main list**
* **list.filter(func(){});** // returns a new list

list.filter(function(item){

return item % 2 == 0

}

اگر **true** بشه اون آیتم ها رو بر می گردونه یعنی بر اساس شرط فیلتر می کنه

* **list.map(func(){});** // returns a new list

اون **عملیات** رو انجام می ده روی تک تک آیتم ها و به شکل یک لیست بهمون بر می گردونه

list.map(function(item){

return item \* 2

}

**Strings Ecmascript5**

* **str[index]** // returns the char
* **str.charAt(index)** // returns the char
* **str.length**
* **str.concat(string2)** // returns a **new** combined string, it doesn't affect the original strings
* **str.toUpperCase()** // returns the string in uppercase, does not effect the original string
* **str.toLowerCase()** // returns the string in lowercase, does not effect the original string
* **str.split(char)** // returns a list of substrings splitted by the char
* **str.trim()** // it wipes all the excess spaces, it doesn't effect the original strings
* **str.indexOf(substring)** // it retruns -1 if it doesn't find the substring in the string
* **str.substr(strartIndex,endIndex)**
* **str.replace(regexp or string, string2)** // It searche the str for the first parameter and if found replaces it with string2.

**Objects**

* **{ }** // to keep data in the form of key/value
* **Key/value** : the value of keys can be a list, integer, string, object, function

var obj = {

key1 : 2,

'key-2' : 'Hi' ,

'Detail' : {

'Data1' : [1,2,3],

'Data2' : ''

},

'fn' : function (){

return (this.key1 \*2); // this refers to obj

}

}

* **obj.key or obj[key**] // To add to or access and change (get and set) the properties, for example: obj.fn() or obj["fn"]()
* **obj["key-1"]** // They have the same naming rules as variables, but if you want to use a dash in the key names, use " " and access it by [ ]
* **this.key** // to access the properties inside the obj
* Another way of defining an object in eccma6 is with **new Object()**

var obj = { }

var obj = **new Object()**

Object.keys(obj) ; // returns an array of the keys

**for-in**

* It is used to traverse objects

let obj = {

name: 'foo',

age : 44

}

**for**( let key in obj ) {

console.log(**key,obj[key]**) // obj.key does not work because key is a variable

}

**Constructor functions**

Functions can be treated as objects.

When you call a function in the usual way, it doesn't recognize ***this*** keyword. But when you create an object from that function, ***this*** can be accessed. Because objects are alive in memory and we have to distinguish them and their properties from eachother.

function func(){

this.prop1 = 'val' // default value

this.prop2 = 5

this.func = function(){

}

}

var obj = **new func**() // obj is an object by the name of its constructor function, here func

console.log(obj.prop1)

* Why should we use them? Because we can make different objects.

function func(prop1, prop2){

this.prop1 = prop1

this.prop2 = prop2

this.func = function(){

}

}

var obj = **new func**('val',2);

var obj = new func('val2',3);

function Person(name,family) {

    this.name = name;

    this.family = family;

    this.find\_the\_length = function () {

        // namepars

        return this.name.length;

    }

}

var person1 = new Person('Sarah', 'Pitt')

var person1 = new Person('Jackaline', 'Scott')

console.log(person1.find\_the\_length())

* instanceof // **person1 instanceof Object** returns true, instanceof checks if smth is an instance of something else.
* person1.cunstructor // returns the constructor of that object, aka the definition of the Person const function
* { } is also using an object constructor which is native to javascript core codes and is interpreted through browser javascript engines which have the native codes like the Object constructor function (var obj = new Object();).

**Prototypes**

* Prototypes are objects that let us access the properties of an object, which we create or are defined in native codes.
* When it comes to inheritance, JavaScript only has one construct: objects. Each object has a private property that holds a **link** to another object called **the prototype**. This prototype object also has a prototype, and this continues until we reach an object whose prototype is set to null. By definition, null has no prototype and acts as the final link in this **prototype chain**.
* Nearly all objects in JavaScript are instances of [**Object**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object) which sits on the top of a prototype chain.
* The methods and properties are **not** copied from one object to another in the **prototype chain**. They are accessed by walking up the chain through links. It checks the constructor function; if methods and properties are not found there, it checks the constructor function's prototype and goes forward in the prototype chain.
* Inheritance is based on prototypes. We can access the properties of the primary Object constructor because when we define our object, there is a prototype for our constructor function, and there is a prototype for the primary Object constructor that is given to all objects.
* Every object we create has a main prototype that includes these main objects
  + The constructor function that we declare and can override the properties of the main Object constructor.
  + Methods we define as a prototype for the constructor function (not as the function methods), using keyword prototype

Person.prototype.fun = function() {

return 'fun';

}

* + the native code Object constructor and all its properties are inherited
* Using keyword prototype, we can add properties whenever we want to any constructor, even the main Object constructor. All objects that we make out of that constructor have those properties.
* Note: In general, extending the prototype of **native classes** is a bad practice.

Array.prototype.newmethod() = function() {} //Don't

* new List() and new String() constructors. They are all objects that inherit the main Object constructor properties. It's the same as using [ ] and " ".
* Look at the prototype chain in the browser console section.

**Inheritance**

**obj from obj**

* Usually it's not used in frontend.
* **var obj = Object.create(null)** // creates an object with no prototype.
* **obj.\_\_proto\_\_** // shows the prototypes of the object
* **Object.create(obj)** // inherits from obj, its properties and values

var car4 = Object.create(car3);// car4 has the same properties and

values as car3

// You can define new properties

car4.makeASmile = function() {

    console.log(':)')

}

* Important: All the inherited parts from parents are prototypes, while overridden or newly defined properties are the usual properties.

var car4 = Object.create(null)

car4.makeASmile = function() {

    console.log(':)')

}

var car5 = Object.create(car4);

car5.color = 'Yellow'; // not prototype

car5.year = 2010;

In browser inspect: car5

1. *{color: 'Yellow', year: 2010}*
   1. color: "Yellow"
   2. year: 2010
   3. *[[Prototype]]*: Object
      1. makeASmile: *ƒ ()*

**Bind, Apply, Call methods**

* They send an object to an extranl function or method and use **this** keyword to access the objects' properties. We can use arguments for that function and pass the properties like that, but in specific circumstances, we want to conveniently read the properties of an object inside another function by using the keyword **this**. We can achieve it by using these methods. They are primarily alike and have minor differences.

**Apply**

* func.apply(obj) // we don't call the function with parentheses; apply firstly processes obj and makes it accessible in func and then calls it.
* func.apply(obj,**[**arg1,arg2**]**) // pass other arguments in a list

**Call**

* Similar to apply
* func.apply(obj, arg1, arg2) // pass other arguments with comma

**Bind**

* after passing the obj, it doesn't call the function automatically; we have to call it

var f1 = func.**bind**(obj, arg1, arg2);

f1();

func.**bind**(obj, arg1, arg2)()

* We can also use them to pass an object to the methods of different objects.

var car = {

make : 'sipa',

model : '111',

displayDetails : function(year , name) {

console.log(year , name)

console.log('Maker : ' + this.make + ' , Model : ' + this.model)

}

}

var car2 = {

make : 'sipa',

model : 'tiba2',

}

car.displayDetails.apply(car2, [2012 , 'hesam'])

**Debugging**

* Using console.log()

    console.log('Start javascript');

* Debugger statement; If no debugging is available, the debugger statement has no effect.

    debugger;

* Debuggers in editors
* Chrome, dev tool

**Visual** **studio**

* In the debugger setting, chose chrome launch
* For only HTML files in **launch.json** instead of url we use this option:

      //We don't need this

       //"url": "C:/Users/Alma/Desktop/Git/Test",

       //Add this:

       "file": "${workspaceFolder}/index.html",

* You can see the value of variables by hovering over them
* Using the watch section in the visual studio debugger, you can track variables by adding their names to the watch list. Note that you should have a breakpoint to see the variables. In Babel, for example, the variables are in the bundle file.

**Chrome**

* In the console tab, in the dev tool, you can see the file and the line, example: test.js:17
* In the source tab in the dev tool, set breakpoints and then reload the page.

**Callbacks**

[are-javascript-callbacks-just-anonymous-functions-sent-as-an-argument-in-a-funct](https://stackoverflow.com/questions/23882149/are-javascript-callbacks-just-anonymous-functions-sent-as-an-argument-in-a-funct)

JavaScript callbacks are functions passed as values into an asynchronous function for the purpose of continuation.

A function has another function as a parameter when something is achieved; it executes that call back function itself. It calls it and runs it like eventListeners callbacks.

**Functions are values:**

So in JavaScript, you can pass functions around as values. You can reference a function in several ways:

* Pass a literal **anonymous** function as the callback

doSomeWork(**function (err, result) {**

**if (err) {**

**throw new Error(err);**

**} else {**

**console.log(result);**

**}**

**}**);

* Pass a **named** function as the callback (Naming every function is a smart idea because you can see it in the stack trace)

doSomeWork(function **magicalCallback**(err, result) {

if (err) {

throw new Error(err);

} else {

console.log(result);

}

});

* Pass in the value of a **variable** that happens to be storing a function as the callback

var **someFunction** = function **callItWhatYouWant**(err, result) {

if (err) {

throw new Error(err);

} else {

console.log(result);

}}// reference the callback stored in the someFunction variable

doSomeWork(**someFunction**);

* Pass in the function by referencing the **function** **name** as the callback

function **callItWhatYouWant**(err, result) {

if (err) {

throw new Error(err);

} else {

console.log(result);

}}

// reference the callback function using the function name

doSomeWork(**callItWhatYouWant**);

* You can't use a value out of the function call after the callback, as the function returns before the callback. You can return whatever you like from the function and use that as normal, but as that happens before the callback, you don't have access to the result from the asynchronous call at that time.

function fun (\_calltback){

\_calltback("Derry")

}

let outp = **fun(**function t(data){

console.log('hi\* ' + data)

**return** ('hi' + data)

}**)** // **fun** is basically done before callback returns so you can't access what is returned from the callback.

console.log('normal procedure: ' + outp)

// output:

// hi\* Derry

// normal procedure: undefined

function fun (\_calltback){

\_calltback("Rose")

return "Derry" // You can return something not related to callback in normal way

}

let outp = fun(function t(data){

console.log(data)

return (data)

})

console.log('normal procedure: ' + outp)

// output:

// Rose

// normal procedure: Derry

**Try/catch/finally**

* Try some code, and if an error has occurred, catch it
* The error that is caught is an object with two properties, error.name, and error.message.
* Throw keyword. In a certain context, the throw is used to create custom errors and then send it to catch. It accepts Boolean, numbers and strings, and objects. We can create our own properties for this object so we can do whatever we want with it inside catch.
* Finally is always executed after the try/catch
* Common errors: syntaxError, refrenceError, typeError

**Timer/interval**

* setTimeout(function (){ do what you want},time in millisecond) // this method has two parameters, Do something after a certain time. It returns an id you can use **clearTimeout(id)** to stop it.
* setInterval(callbackfunction, time)// Do something periodically. The parameters are the same as timeout. It returns an id. You can clear the ineterval by **clearInterval(id)**

let intervalId = setInterval(function(){

if(condition){

clearInterval(intervalId)}

},2000)

* We write websites sliders with intervals

**Date**

* Mozilla.org, date section, [Date](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date)
* It's better to store time in The Gregorian calendar in DB. You can use packages to show it in other calendars.
* new Date()
* Unix Timestamp standard is a method that defines the time in milliseconds from the first day of **1970**. You can store these milliseconds and then use the language API to convert them to the usual format. Past or present or future.

Var d = new Date();

d.**getTime**()// returns 1634552143579 for example

var d2 = new Date(1634552143579 + 1000)// add one second, it returns the usual date format

* Date.now() // now in **miliseconds** Using Date objects, it's a static method.
* Date.parse(year/month/day) // returns time in **milliseconds**
* Set and get methods, getters, and setters, for Date

**Math** [Math](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math)

* Math.PI // returns the p number
* Math.abs(number) // returns the absolute of the number, returns a positive or zero form of the number, if the number is 'null' it returns 0
* Math.floor(number) // returns the largest integer less than or equal to a given number.
* Math.random(number) // returns a number between 0 and 1: 0.6031627200095537

// It returns random numbers from 0 to max (does not include max)

// Because even if you have (0.9999999999999 \* max) and then floor it it'll be less than max, it is never 1 to return the max 1\*max = max

Math.random()

function getRandomInt(max) {

    return Math.floor( Math.random() \* Math.floor(max))

}

* Math.round(number) // If the decimal is 0.5 or more, it gives the next bigger integer number and if it is less than 0.5, it gives the previous integer number
* Math.pow(number, pow)

**DOM (Document Object Model)**

* document is an **object**, having all the tags of the page, to make the page elements dynamic. **DOM** is this object.
* All of those children are also objects

**Select**

* **document.children**, returns the children of the document which has the html too, they are objects with their methods. html-> (head ->(link, meta), body->div)
* document.**head**, document.**body** // accessing special tags
* document.**getElement(s)By**

**Id**

**TagName**

**ClassName**

**Name**

* document.**querySelector**('.list.green') // returns **one** item with class list that also has class green
* document.**querySelectorAll**('.list'),('li'),('#list') // it returns a list that we can loop through with forEach
* document.**querySelectorAll**('ul li')**[0].**style
* document.body.**firstChild** // usually returns text of that element
* document.body.**firstElementChild**
* el.**nextSibling**
* document.body.firstElementChild.**nextElementSibling**
* el.**parentElement**
* el.**closest**(".list-item") // finds the closest ancestor to el with list-item class

**Altering and getting attributes**

* document.**body**.**children[0].style.**color =""
* document.body.children[0].**textContent** = ""
* document.body.children[0].**innerHTML** = "" //**Important: HTML is all in capital**
* document.body.children[0].**classList.add()** or **remove()** = ""
* el.**setAttribute**("style", "background-color:blue") // set the value of an attribute of the element
* el.**getAttribute**("data-x") // get the value of an attribute of the element
* el.**setAttribute**('id', 'value')
* el.**id** = 'value'

**Adding and remove elements to DOM**

* document.**createElement**('ul')
* el.classlist.**add**('class-name')
* el.**toggle**('class-name') // adds the class if it does not exist, removes it if it exists
* el.**appendChild**(el2)
* el.**after**(el2 or text) // adds el2 to the after of the el. el and el2 have the same parent
* el.**remove** // removes the element
* el.**removeChild**('ul') // removes the child and returns it
* el.**insertBefore**(newChild, existingChild) // it adds new child to before of the newChild

**Dialog**

* It's better that we don't use the default javascript dialogs because they don't have a good style.
* Sweetalert is a package among packages with this purpose that we can use instead. It uses HTML to create dialog.
* alert () function // It's related to the browser. You can't change it visually that much. At first, it was used for logging.
* confirm(message) // returns true or false. In the past was used to ask the user if they want to quit a form
* prompt(message) // It has an input to fill and return its value.

**Events** [Events](https://developer.mozilla.org/en-US/docs/Web/API/Event)

* both server-side and client-side

**EventHandler**

* If you look at the prototype in an el, you see some **onX** (onclick, for example)methods. These are the methods that work with events as eventHandlers. They are mostly empty on default.
* Some of them are specific to elements, like onchange in inputs.
* el.onclick = function () { } // we can add a function to that **eventHandler** for that element. Like defining function with var x = function() {}. It runs el.onclick()
* browser implement these handlers differently which you have to consider
* The event happens and it is given to the handler.

el.onclick = function (**event**) { consol.log(event)} // returns the mouseclick object

el.onclick = function (**event**) { event.target.style.background = 'green'; console.log(event);} //target is what is clicked

let title = document.querySelector('#title')

title.onmouseenter = function(event) {

event.target.style.backgroundColor = 'red';

}

title.onmouseout = function(e) {

e.target.style.backgroundColor = 'yellow';

}

* We can't have more than **one** event handler

**EventListener**

* This is a better method
* We can have more than one event listener for the same el.
* el.**addEventListenere**(event type, callback function)

el.addEventListener( 'click', function(**event**) {

// code1

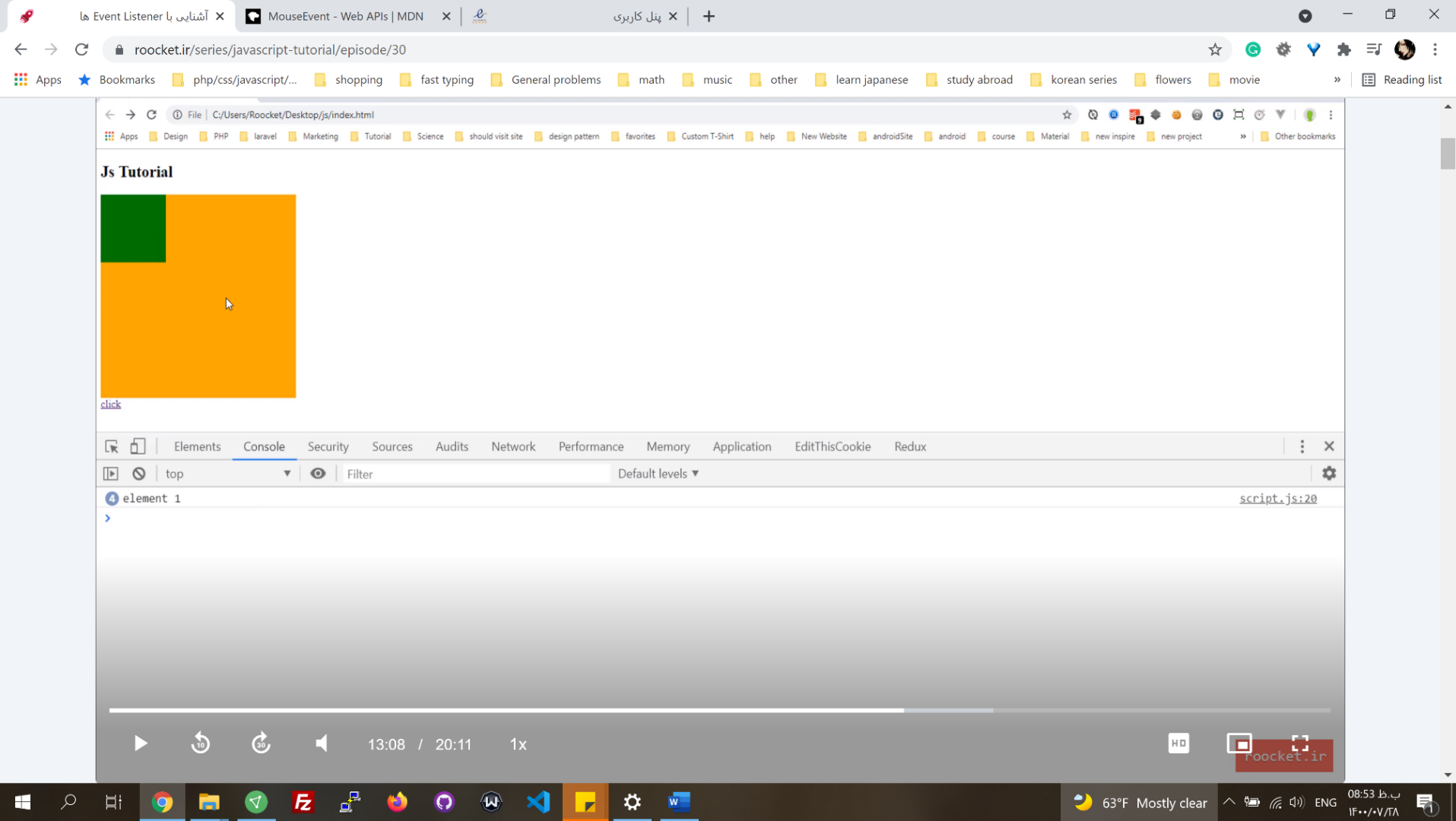
});

el.addEventListener( 'click', function(**event**) {

// code2

});

* event.**stopPropagation**() // If we have eventListeners both on the child and the parent, the event on child triggers both the child callback and the parent callback. Sometimes this can cause a problem. It can be prevented. You can handle this using **event** object passed to the callback function. It is an object, having methods and properties. One of them is **stopPropagation()** that prevents the shared event triggered on the child to trigger the event on the parents.



parent

parent

title.addEventListener( **'click'**, function(event) {

event.stopPropagation() // click on the parent wont be triggered

// code

}); //title is the child, parent also has a click addEventListener

|  |  |  |
| --- | --- | --- |
| 0 | UNSENT | Client has been created. open() not called yet. |
| 1 | OPENED | open() has been called. |
| 2 | HEADERS\_RECEIVED | send() has been called, and headers and status are available. |
| 3 | LOADING | Downloading; responseText holds partial data. |
| 4 | DONE | The operation is complete. |

// code

});

* event.**preventDefault** // prevents event **default behavior** like clicking on links or submitting on the form. Sometimes we need that. For example, when you create a link for submitting a review, you only need to open a modal and work on that. In forms too, when you use ajax, you use this to prevent sending info to another route.
* el.**removeEventListener**( event type, callback function) // To remove listeners, you have to create a name for the callback functions, so you specify what you want to remove. We can put the callback in a var.

**HttpRequest and** **Ajax** [XMLHttpRequest](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest)

* The pure javascript has a constructor function to send and receive data from a URL. You can define an object by this constructor function.
* XMLHttpRequest // It's a built-in object in browsers, so it must run on browsers
* HTTP Methods: GET, POST, PUT, HEAD, DELETE, PATCH, OPTIONS
* XMLHttpRequest.readyState, 0, 1, 2, 3 ,4

|  |  |  |
| --- | --- | --- |
| 0 | UNSENT | Client has been created. open() not called yet. |
| 1 | OPENED | open() has been called. |
| 2 | HEADERS\_RECEIVED | send() has been called, and headers and status are available. |
| 3 | LOADING | Downloading; responseText holds partial data. |
| 4 | DONE | The operation is complete. |

* Status: An HTTP **status code** is a **server response** to a browser's request.

|  |  |
| --- | --- |
| HTTP Status Code 200 | Get request OK. |
| HTTP Status Code 201 | Post request OK. |
| HTTP Status Code 301 | Permanent Redirect. |
| HTTP Status Code 302 | Temporary Redirect. |
| HTTP Status Code 404 | Not Found. |
| HTTP Status Code 410 | Gone. |
| HTTP Status Code 500 | Internal Server Error. |
| HTTP Status Code 503 | Service Unavailable. |

**Get**

* Receiving data from a URL

var xhttp = new **XMLHttpRequest**();

// eventHandlers onreadystatechange, we can use just once

**xhttp**.**onreadystatechange** = function() {

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

// Typical action to be performed when the document is ready:

document.getElementById("demo").innerHTML = xhttp.responseText;

// plain text

} };

xhttp.**open**("GET", "filename", true);

xhttp.**send**();

var xhttp = new XMLHttpRequest();

// eventHandlers onreadystatechange

xhttp.onreadystatechange = function() {

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

// Typical action to be performed when the document is ready:

document.getElementById("demo").innerHTML = xhttp.responseText;

}

};

xhttp.open("GET", "filename", true);

xhttp.send();

**Post**

* Send data to a URL
* xhttp.setRequestHeader(header, value) // Headers specify what kind of data type we are sending

var xhr = **new** **XMLHttpRequest**();

xhr.**open**("POST", '/server', true);

// Send the proper header information along with the request

xhr.**setRequestHeader**("Content-Type", "application/x-www-form-urlencoded");

// XMLHttpRequest.DONE = 4

// event listener, can have many listeners.

xhr.**addEventListener**(**'load'**, function() { // Call a function when the state changes.

if (this.readyState === XMLHttpRequest.DONE && this.status === 200) {

// Request finished. Do processing here.

}});

// the data sent is in a format that server accepts.

xhr.**send**("foo=bar&lorem=ipsum");

// xhr.send(new Int8Array());

// xhr.send(document);

// xhr.send(JSON.stringify(data));// to send json

* JSONPlaceholder is a website for test API services. [https:// jsonplaceholder.typicode.com/users](https://jsonplaceholder.typicode.com/users)
* To see the requests in the dev tool in chrome, look at the network tab.,
* The pure XMLHttpRequest is not suggested, because it's troublesome and the code is very long. So it's better to use libraries.

**Transition event**

To control transition behavior on elments that have transition in their css. May not be supported in some browsers.

* transitionrun
* transitionstart
* transitioncancel
* taransionend

El.addeventListener("transitionend", function)

**Libraries**

* JQuery // API methods to work with its core, single page application
* Axios Ajax Specific library
* React js // Creating the user interface
* React router

**Frameworks**

* Putting together a few libraries
* Vue js // virtual DOM, single-page apps, can be used for normal websites.
* Angular // Only used to create **single-page applications**
* alpinejs // A rugged, minimal framework for composing JavaScript behavior in your markup

<template **x-for**="post in posts">

<h2 x-text="post.title"></h2>

</template>

* React Bootstrap. React-Bootstrap replaces Bootstrap JavaScript.