



JavaScript poate fi folosit si ca backend, cu node.js

**Adaugarea la script**

* Putem pune scriptul si intre <body> </body> si in <head> </head>, asa:

<script src="script.js"> </script>

Mereu folosim <script> </script> ca aici va fi inserat scriptul

**Conventii**

* JS tot foloseste camelcase. Prima litera din variabila e lowercase, si fiecre nou cuvant e uppercase

let firstName

* constantele se scriu toate cu uppercase doar si cu \_

**Data Types**

* Orice valoare in JS este un obiect sau data primitiva
* Orice nu este un obiect, este o primitiva
* JavaScript **are dynamic typing**. Asta inseamna ca nu trebuie sa definim tipul la variabila, ci se determina automat
* O variabila poate stoca un tip de date, apoi sa-i fie asignat cu totul alt tip.
* Tipuri de date:

- **Number** – e folosit pentru decimal si int. Nu exista float, double sau altele, doar number pentru orice numar

- **String –** e pus intre ‘’ sau “”

- **Boolean**

- **Undefined** – variabila neinitializata e de tip undefined

let variabila;

- **Null –** variabila declarata doar ca fara valoare inca

**- Symbol –** valoare unica care nu poate fi schimbata sau obiecte unice constante

**-** **BigInt** – numere mai maru ca Number

* JavaScript nu permite redeclararea variabilelor cu let

let test = true;

console.log(typeof test);

let test;



**Comments**

// a commentn

/\*

first comment

second comment

\*/

// - pe o linie

/\* \*/ - mai multe linii

**typeof operator**

* Arata tipul unei variabile

let test = true;

console.log(typeof test);

console.log(typeof 23);

Si ne va afisa: boolean

**let, const and var**

* **let** – este block scoped. variabile care pot avea referinte catre orice tip de date, si tipul lor poate fi modificat oricand

let test = true;

test = 100;

* **const** – variabila trebuie neaparat initializata la declarare si nu mai poate lua alta referinta!

const test;



const TEST = 10;

TEST = 100;



* **var(de evitat!) –** este function scoped.
* **global scope:**

function test() {

if (true) {

a = 100;

}

console.log(a);

}

test();

console.log(a);

Vedem ca a nu are nici const, var sau let, deci e declarata ca variabila globala si poate fi folosita oriunde in program, indiferent unde e declarata.

**let vs var**

* **let** este block scoped, **var** este function scoped. Deci, let este disponibil doar in blocul definit, if() { }, dar var este disponibil in intreaga functie, adica indiferent unde e declarata, ea apartine la blocul functiei

function test() {

if (true) {

let a = 100;

}

console.log(a);

}

test();

ERROR: a is not declared

function test() {

if (true) {

var a = 100;

}

console.log(a);

}

test();

**Dar atentie! var face parte din scopul functiei oriunde e declarat in ea!**

function test() {

if (true) {

var a = 100;

}

console.log(a);

}

test();

console.log(a);



* **hoisting** e permis cu **var**, dar nu si cu **let**

console.log(a);

let a = 100;

ERROR!!!

**Hoisting**

* Concept din JS care permite apelarea functiilor sau a variabilelor inainte ca ele sa fie definite in ordinea lor
* In JS, Hoisting face ca indiferent unde declaram functiile sau variabilele in code, ele mereu vor fi initializate in scope a lor inainte de a executa codul propriu zis

test();

test();

function test() {

if (true) {

a = 100;

}

console.log(a);

}

Declararea functiei se va face inainte de apelarea ei, indiferent de ordinea in care e scrisa in code;

* Variabilele declarate cu var, desi sunt declarate inainte de a executa codul,ele nu sunt si intializate atunci!

console.log(a);

var a = 100;

Undefined

**Operators**

* \*\* - putere
* typeof
* Operatorii matematici(+,- ...) mereu se executa inaintea la cei de comparare
* Executia operatiilor mereu se face de la dreapta la stanga:

a = b = 100 + 50;

1. 100 + 50
2. b = 150
3. a = b

**Template Literals**

* Folosirea la + nu e comoda pentru a concatena stringuri
* E mult mai usor sa folosim `. Atentie, nu qutoes, ci `, deasupra la TAB
* cu ${} putem face referinta la variabile si chiar face operatii in el, gen ${var1 + var2}
* Totodata, el deodata vede newline si le pune automat, fara \n

let *name* = 'Eduard';  
let *birthYear* = 2002;  
let *job* = 'Java Developer';  
  
*console*.log(  
 `My name is ${*name*}  
 I'm ${2023 - *birthYear*}  
 And my job is ${*job*}`  
)

**Type Conversion and Type Coercion**

* **Type Conversion** – conversia manuala a unui dip de date in altul
* **Type Coercion** – conversia automata a unui tip de date in altul

De ex, un user introduce niste date intr-o forma, cum ar fi anul nasterii. Vrem sa calculam anul cand el va fi major. Problema e ca datele de la forma vin in format de string, deci o asa operatie:

const *birthAge* = '1991';  
*console*.log(*birthAge* + 18);

Ar rezulta: ‘199118’

Se va face concatenarea la String, nu adunarea, caci ‘1991’ e String

* Aici ne poate ajuta Type Conversion, si anume functia **Number()** ce ia un string si il transforma in numar
* const *birthAge* = '1991';  
  *console*.log(Number(*birthAge*) + 18);
* Daca oferim nu un numar la Number(), vom primi **NaN** – not a number
* Functia **String()** transforma in string
* **Evident, putem doar face conversii la Boolean, String si Number**
* Type Coercion se produce in situatiile cand avem un string + Number:

const *birthAge* = '1991';  
*console*.log(*birthAge* + 18);

Si primim 199118

Asta e valabil in cazul la String + number

Daca avem String – number sau String – String, string deja e transformat in Number:

const *birthAge* = '1991';  
*console*.log(*birthAge* - 18);

1973

* Deci:

String + Number => Number in String

String [-/\*] Number => String in Number

String – String => Strings in number

*console*.log('10' - '5');

5

**Convert to Boolean**

* **Boolean()**
* **False** sunt automat:

- 0

- ‘’

- NaN

- undefined

- null

* Orice altceva este **True**

**== vs ===**

* === - este Strict Equality operator. El nu compara doar valoarea, ci si tipul la date sa fie acelasi. Deci, el nu face conversii, ca ==
* De ex:

*console*.log('10' – '5');

true

*console*.log(18 === '18')

false

* === e preferat, caci == poate cauza surse de erori din cauza la conversie

**!= vs !==**

* != - verifica daca valorile sunt diferite, dar face conversiile
* !== - nu face nicio conversie

**prompt(“”)**

* Functia data face sa se deschida in pop in pagina, sa ceara niste informatii si sa returneze string pe carel- userul l-a scris

let *input* = prompt("Enter your age");  
*console*.log(*input*);

**Switch**

* Foloseste ===, nu ==

**Ternal operator**

* Operatorul ternal poate fi folosit si pentru a returna valori
* Asta merge si in Java!

let *age* = 23;  
const *drink* = *age* >= 18 ? "wine" : "water";  
*console*.log(*drink*);

**Strict mode**

* Strict mode face codul mai securizat
* Il activam scriind asta la inceput de document:

'use strict';

* El face ca sa fie aruncate mai multe exceptii, facand codul mai securizat si eliminand posibilele buguri
* Beneficiile sale:

- Unele erori silentioase, pe care JS nu le afiseaza, arunca execptii

- Fixeaza greseli care fac ca JS engines sa ruleze optimizari, deci asa strict mode poate uneori face ca codul sa ruleze mai repede

- Nu permite folosirea numelor de variabile a posibilelor chestii ce ar putea fi puse in viitor. De ex, asa nume la variabila e invalid cu strict mode doar:

let interface;



**- Nu permite accesul la global scope, facandu-l inaccesibil, deci permite definirea variabilelor folosind doar const, var si let!**

**- Nu permite folosirea la delete keyword pentru variabile, ci doar pentru fields:**

'use strict'  
let *a* = 100;  
delete a;



Dar permite stergerea la fielduri

'use strict'  
let *a* = {message : "test"}  
delete *a*.message;



**- Nu permite dublicarea parametrilor**

'use strict'  
  
function test(a,a){  
   
}



* with statment este interzis
* Strict mode poate fi folosit fie in intregul script, fie doar in functii.

- In totul script: punem ‘use strict’ la inceput de document

- In functii: punem ‘use strict’ exact dupa definirea functiei

function a(){  
 'use strict'  
 let interface;  
}

* Daca o functie e apelata din interiorul la o metoda, functia nu va avea acces la this, si va fi undefined, desi nu asa va fi si fara strict mode
* 'use strict'  
    
  const *person* = {  
   name: 'Eduard',  
   getName: function() {  
   function test(){  
   *console*.log(this);  
   }  
   test();  
   }  
  }  
    
  *person*.getName();

undefined

**Functions**

* Parametrii functiei nu folosesc let sau var!

function test(a, b){  
}

* Daca o functie nu are niciun argumnt, si trimitem unul, el nu va fi folosit

function test(){  
}  
test(100);

* Daca nu folosim return, orice functie returneaza in mod default undefined
* **Functiom declaration**: cream o functie cu nume

function test(){  
}

Avantajul e ca o putem apela pana ea sa fie declarata, asa oricum JS intai o va defini si apoi va executa codul

*console*.log(test(21));  
function test (birth){ return 2023-birth };

* **Function expression**: cream o functie fara nume, dar o stocam intr-o variabila. Asta e numita **Anonymus Function**
* let test = function (birth){ return 2023-birth};  
    
  *console*.log(test(21));

**Nu o putem apela pana ea sa fie declarata**

*console*.log(test(21));  
let test = (birth) => 2023-birth;



* **Arrow function** – nu necesita function key word, dar foloseste () => { }

let test = () => {}

Daca avem un singur parametru, () nu e necesar

Tot sunt **anonymous function!**

**Mai sunt numite Lambda Function**

**Arrow function nu ofera acces la this, daca nu sunt executate din interiorul unei metode!**

* Putem oferi si mai putini parametri actuali decat formali, dar cei neoferiti for fi undefined

**Arrays**

* Se folosesc [], nu { }
* Un array este un obiect
* Putem sa il cream si cu new Array(...)

const *array1* = [1,2,3,4,5];  
const *array2* = new *Array*(1,2,3,4,5);

* Putem stoca diferite tipuri de date intr-un array:

const *array1* = [1,2,3,'Hi',true];

* Array in JS este un obiect mutabil, deci nu trebuie sa definim size a lui, si putem adauga oricate vrem.
* **Proprietati la array:**

.**length** – contine numarul de elemente din array

* Metode la obiectul array:

- **.push(el1, el2, ...)** – adauga elemente la final de array si returneaza noul length la array

- **shift**() – sterge elementul la inceput d array

- **.unshift(el1, el2, ...)** – adauga elemente la inceput de array si returneaza noul length la array

- .**pop()** – sterge ultimul element din array si il returneaza

- **.indexOf(element) –** returneaza indexul la element sau -1 daca nu il gaseste

- **.includes(element)** – true daca element e in array si false daca nu

- **splice(start, nr)** – sterge un numar de nr elemente incepand cu start, inclusiv start

const *array* = [1,2,3,4,5];  
*array*.splice(1,2);  
*console*.log(*array*);

[1, 4, 5]

- **splice(start, nr, el1, el2, ...)** – sterge un numar de nr elemente incepand cu start si apoi adauga elementele el1 si el2 incepand cu start

const *array* = ['element1','element2','element3','element4','element5'];  
*array*.splice(1,2, 'new1','new2');  
*console*.log(*array*);

[ 'element1', 'new1', 'new2', 'element4', 'element5' ]

putem folosi 0 ca nr pentru a nu sterge nici unul si pur si simplu sa inseram incepand cu un index

const *array* = ['element1','element2','element3','element4','element5'];  
*array*.splice(1,0, 'new1','new2');  
*console*.log(*array*);

[

'element1',

'new1',

'new2',

'element2',

'element3',

'element4',

'element5'

]

* **slice(from, to)** – returneaza un nou array ce contine elementele de la from pana la to, from inclusive,to exclusive

Daca nu folosim from, se pune 0 automat, adica slice(5) inseaman de la 0 la 5, 5 exclusiv

Putem folosi si indexi negativi, gen slice(1,-2) – de la 1 pana la penultimul element, penultimul exclusiv. Decim -1 e ultimul, -2 penultimul si tot asa

* **some(functie)** – returneaza true daca cel putin un element din array corespunde conditiei din functia predicat

const *array* = [10,20,30,40,51]  
  
*console*.log(*array*.some((el) => el % 2 !== 0))

true

* **reduce((total, current) => { }, initial)**

- primul argument e o functie, ce ia total, adica rezultatul carat de la o reducere la alta si current care e elementul current

- initial e optional. Daca nu e oferit, se iau primele 2 elemente pentru a face un total, apoi total e tarait la fiecare operatie, si se ia cate un element doar. Daca e oferit, nu se iau primele 2 elemente ca prima operatie, ci primul si acest initial ca total

const *array* = [10,20,30,40,50]  
  
*console*.log(*array*.reduce((total, el) => total+el))

150

intai 10 + 20

apoi 30 + 30

apoi 60 + 40

apoi 100 + 50

const *array* = [10,20,30,40,50]  
  
*console*.log(*array*.reduce((total, el) => total+el,1000))

1150

* **at(index)**
* **concat(arr)** – creaza un nou array ce contine elementele cei curent + cele a lui arr puse la urma

const *array1* = [10,20,30,40,50]  
const *array2* = [60,70,80];  
  
*console*.log(*array1*.concat(*array2*));

* **every(predicate)** – verifica daca toate elementele corespund la predicate

const *array1* = [10,20,30,40,50]  
  
*console*.log(*array1*.every((el) => el<100));

* **fill(value, start)** – inlocuieste value incepand de la start in array
* const *array1* = [10,20,30,40,50]  
  *array1*.fill(0,2);  
    
  *console*.log(*array1*);

[ 10, 20, 0, 0, 0 ]

* **filter(lamda)** – creaza un nou array doar cu valorile care corespund filtrului

const *array1* = [10,25,30,45, 60, 81]  
  
*console*.log(*array1*.filter((el) => el % 2 === 0));

* **find(lambda)** – gaseste primul element care corespunde expresiei lambda

const *array1* = [11,25,30,45, 60, 81]  
  
*console*.log(*array1*.find((el) => el % 5 === 0));

* **findIndex(lambda)** – returneaza primul index al elementului ce corespunde prediacutului
* **findLastIndex(lambda)**
* **flat(level)** – transforma un array cu mai multe arrays in el in array cu elemente, fara arrays in el. Level arata cat de adanc sa faca concatenarea. Gen, 1 va face ca sa se concateneze [] cu [], 2 [] cu [, [], ] si tot asa

const *array1* = [1,2, [3, 4, [5, 6 , 7]]]  
  
  
*console*.log(*array1*.flat());

[ 1, 2, 3, 4, [ 5, 6, 7 ] ]

const *array1* = [1,2, [3, 4, [5, 6 , 7]]]  
  
  
*console*.log(*array1*.flat(2));

[

1, 2, 3, 4,

5, 6, 7

]

* **forEach(lambda)**
* **join(delimiter)** – creaza un string cu elementele din array, ele fiind delimitate prin delimiter

const *array1* = [1,2, [3, 4, [5, 6 , 7]]]  
  
  
*console*.log(*array1*.join("|"));

* **reverse()** – afecteaza array original

const *array1* = [1,2,3,4,5,6,7,8,9]  
*array1*.reverse();  
  
*console*.log(*array1*);

* **sort(comparator)** – sorteaza pe baza la un comparator function. Exact ca la Java. Sorteaza array original si returneaza o referinta la el
* const *array1* = ['aa','gfgfg','d','dfdfd','dfd'];  
    
  *console*.log(*array1*.sort((el1, el2) => {  
   if(el1.length > el2.length)  
   return 1;  
   else if(el1.length < el2.length)  
   return -1;  
   else return 0;  
  }));
* **map(lambda)** – returneaza unul nou, nu il afecteaza pe cel curent.

**Objects**

* Metoda mai veche de a crea obiecte in JS, este prin a folosi { }, adica asa ca in JSON:

const *obj* = {  
 firstName: 'Eduard',  
 lastName: 'Mititiuc',  
 age: 21  
}

* Accesarea unei proprietati a obiectului se poate face in 2 moduri:

- Prin a folosi punctul

*console*.log(*obj*.firstName)

- Prin a folosi [‘field’]

*console*.log(*obj*.firstName)

Prin [] putem folosi si o expression, gen asa:

const *field* = 'firstName';  
*console*.log(*obj*[*field*]);

* In JS, putem adauga fielduri oricand vrem la un obiect. O putem face prin:

- punct:

const *obj* = {  
 firstName: 'Eduard',  
 lastName: 'Mititiuc',  
 age: 21  
}  
  
*obj*.location = 'USA';

- brackets

const *obj* = {  
 firstName: 'Eduard',  
 lastName: 'Mititiuc',  
 age: 21  
}  
  
*obj*['location'] = 'USA';

**Object Methods**

* Pentru a adauga o metoda unui obiect, folosim asa approach:

const *obj* = {  
 firstName: 'Eduard',  
 lastName: 'Mititiuc',  
 age: 21,  
 getFullName: function () { return this.firstName + ' ' + this.lastName}  
}  
  
*console*.log(*obj*.getFullName())

**Mereu accesam fields doar cu this! Daca nu folosim this, nu se va cauta nimic in fields de la object!**

const *obj* = {  
 firstName: 'Eduard',  
 lastName: 'Mititiuc',  
 age: 21,  
 getFullName: function () { return firstName+lastName}  
}



* Putem si asa:

const *obj* = {  
 firstName: 'Eduard',  
 lastName: 'Mititiuc',  
 age: 21,  
 getFullName: () => { return "Test"}  
}  
  
*console*.log(*obj*.getFullName())

**this nu este disponibil in row functions, deci nu putem accesa fields ale object cu row function!**

Apelarea metodei se poate face si prin obj.[‘metoda’]()

*console*.log(*obj*['getFullName']())

* Putem crea un nou field si asa; **this.field**
* const *obj* = {  
   firstName: 'Eduard',  
   lastName: 'Mititiuc',  
   age: 21,  
   getFullName: function () {  
   this.fullName = this.firstName + ' ' + this.lastName;  
   return this.fullName;  
   }  
  }

**For**

for(let i = 0;i<10;i++)  
 *console*.log(i);

* **for in** – returneaza keys a unui obiect

let *array* = [100,200,300,400];  
  
for(let i in *array*)  
 *console*.log(i);

0

1

2

3

Deci, aici keys sunt indexii

let *obj* = {  
 firsName: 'Mititiuc',  
 lastName: 'Eduard',  
 age: 21  
}  
  
for(let i in *obj*)  
 *console*.log(i);

firsName

lastName

age

Deci, am primit numele la fields, nu valorile lor!

* for of – returneaza values a unui obiect ce poate fi interat, ca Arrays, dar nu obiecte!
* let *obj* = {  
   firsName: 'Mititiuc',  
   lastName: 'Eduard',  
   age: 21  
  }  
    
  for(let i of *obj*)  
   *console*.log(i);



let *array* = [100,200,300,400];  
  
for(let i of *array*)  
 *console*.log(i);

**100**

**200**

**300**

**400**