

TEHNICI WEB

JAVASCRIPT

Claudia Chiriță . 2024/2025

JAVASCRIPT

- dezvoltat de **Brendan Eich** în 1995, la Netscape, sub numele de Mocha și LiveScript
- standardizat în 1997 de asociația ECMA (European Computer Manufacturer's Association) sub numele de *ECMAScript*
- versiunea actuală: **ECMAScript 2025**

JAVASCRIPT - WHAT?

- limbaj lightweight de **scripting** pentru pagini web
- combină mai multe paradigme de programare:
imperativă, funcțională, orientată pe obiecte, bazată pe **prototipuri** și pe evenimente

JAVASCRIPT - WHAT?

- limbaj **interpretat** (scriptul este executat direct, fără compilare prealabilă)
- codul rulează în browserul clientului
- single-thread, **dinamic**

JAVASCRIPT - RTM

JavaScript The Definitive Guide, David Flanagan

Understanding ECMAScript 6, Nicholas C. Zakas

JAVASCRIPT + HTML

- o pagină web este reprezentată în memorie ca un arbore de elemente (obiecte)

Document Object Model – DOM

- JavaScript poate interacționa cu documentul HTML prin intermediul DOM-ului

JAVASCRIPT + HTML

- element HTML --- > object JavaScript

```
<h1 id="identif">...</h1>  
ob = document.getElementById("identif")
```

- atribut al unui element HTML --- > proprietate a obiectului JavaScript

```
<h1 id="identif" class="cls">...</h1>  
ob.id, ob.className  
  
ob.src
```

JAVASCRIPT + HTML

- atributul style: proprietăți CSS ---> proprietatea style
-> obiectul style -> proprietăți de stilizare CSS

```
<h1 style="color:pink; text-align:center;">...</h1>  
ob.style.color, ob.style.textAlign
```


JAVASCRIPT @ HTML

codul JavaScript poate fi plasat inline, în head, folosind tagul `<script>`

```
<head>  
<script type="text/javascript">  
/* cod JavaScript */  
</script>  
</head>
```

JAVASCRIPT @ HTML

codul JavaScript poate fi plasat într-un fișier extern, importat în documentul HTML [varianta recomandată]

```
<script type="text/javascript" src="script-file.js"></script>
```

atribute suplimentare, doar pentru scripturi externe:

- **defer** - întârzie executarea codului până se încarcă complet documentul HTML
- **async** permite încărcarea codului asincron față de document

JAVASCRIPT - HOW?

JavaScript este case sensitive și folosește setul de caractere Unicode

- identificatorii
 - denumesc variabile, cuvinte cheie, funcții, etichete
 - formați din: cifre, litere, _, \$;
 - primul caracter: literă, _, \$
- ; separator
- { } bloc de instrucțiuni

-

```
/* comentariu  
pe mai multe linii  
*/  
// comentariu pe o singură linie
```

JAVASCRIPT - WHERE?

JavaScript debugger in browser

<https://jsfiddle.net/>

<https://codepen.io/>

DIALOG: ALERT & PROMPT

un tab al unui browser e reprezentat de un obiect
window (interfața **Window**)

```
alert(mesaj); // afișează o casetă de alertare care
              // conține un mesaj și un buton OK
```

DIALOG: ALERT & PROMPT

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="demo.css">
    <script type="text/javascript">
      /* var x = prompt("gimme gimme gimme a
number!");
      var y = prompt("gimme another
number!");
      alert(typeof(x));
      alert(x+y);
      alert(parseInt(x)+parseInt(y)); */
    </script>
  </head>
  <body>
  </body>
</html>
```

>

DIALOG: ALERT & PROMPT

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="demo.css">
    <script type="text/javascript">
      /* var n = prompt("gimme some number!");
       alert('suma este ' +
suma(parseInt(n)));

      function suma(x) {
        var s = 0;
        for (var i=1; i<x; i++) s=s+i;
        return s;} */
    </script>
  </head>
  <body>
  </body>
</html>
```



CORE JAVASCRIPT

TIPURI DE DATE

- primitive:
 - boolean
 - null, undefined
 - number, bigint, string
 - symbol
- obiecte (tipul **Object**)
obiecte predefinite: **Array, String, Number, Boolean, Math, Date, Set, Function, Error, RegExp**

CORE JAVASCRIPT

INSTRUCȚIUNI

- atribuire: =
- control flow: if...else, switch
- loops & iteration: while, for
- return

CORE JAVASCRIPT

FUNCȚII

```
function nume(param1, param2, ...)  
{  
    corpul funcției  
}
```

CONVERSIE TIPURI

- JavaScript e *dynamically typed*
- nu este necesară precizarea tipului de date, ca în alte limbaje de programare
- tipurile de date sunt convertite automat, la nevoie, în timpul execuției

```
x = 23; //number
y = "abc"; //string
z = true; //boolean
var a; //undefined
cat = {nume:"Tigger", culoare:negru}; //obiect
```

VARIABLE - DECLARARE

- variabilele declarate
 - în interiorul funcțiilor sunt locale pentru acele funcții
 - în afara oricărei funcții se numesc variabile *globale*
 - într-un domeniu (*scope*) sunt accesibile în funcțiile copil
- variabilele trebuie declarate mereu înainte de folosire; JavaScript permitea asignarea de valori unei variabile nedecarate, creând o variabilă globală undeclared - EROARE în strict mode

VARIABLE - INIȚIALIZARE

- se poate atribui o valoare inițială la declarare
- dacă o variabilă este declarată fără inițializare, i se asignează valoarea *undefined*

VARIABLE - DECLARARE - VAR

- declarate explicit folosind cuvântul cheie var

```
var x = 42;
```

- putem declara variabile fie global, fie local la nivel de funcție

VARIABLE - DECLARARE - VAR

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="demo.css">
    <script type="text/javascript">
      /* var x = 0; // var globală
      function f(){
        var y = 1; // var locală
        var x = 5; // var locală
        function g(){
          y++;
          z = 1; // var globală; eroare?
          console.log(y);
        }
        g();
      }
      f(); //va afișa 2
      console.log(x); //va afișa 0
```

>

VARIABLE - DECLARARE - LET

- declarate explicit folosind cuvântul cheie `let`

```
let x = 42;
```

- declară o variabilă locală vizibilă doar în blocul, instrucțiunea sau expresia în care este folosită

VARIABLE - DECLARARE - LET

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="demo.css">
    <script type="text/javascript">
      /* var g = 0; // var globală
       { let g = 1; }
       console.log (g); // va afișa 0
      */
    </script>
  </head>
  <body>
  </body>
</html>
```

>

VARIABLE - DECLARARE - LET

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="demo.css">
    <script type="text/javascript">
      /* function letTest() {
        let x = 1;
        if (true) {
          let x = 2;
          console.log(x); // 2
        }
        console.log(x); // 1
      }
      letTest(); */
    </script>
  </head>
  <body>
  </body>
</html>
```

>

VARIABLE - DECLARARE - CONST

- putem declara constante *block-scoped* folosind cuvântul cheie **const**

```
const x = 42;
```

- se inițializează la declarare
- nu se pot modifica pe parcursul execuției programului prin re-atribuire; nu pot fi redeclarate

VARIABLE - TYPEOF

- variabilele au tipuri dinamice: pot fi declarate (și pot primi valori) cu orice tip de date
- tipul variabilelor nu este specificat explicit, dar poate fi aflat cu

```
typeof (x) ;
```

- tipul unei variabile nedeclarate este *undefined*

VARIABLE - TYPEOF

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="demo.css">
    <script type="text/javascript">
      /*
        x = 42;
        console.log(typeof x); // "number"
        x = "fish";
        console.log(typeof x); // "string"
        x = true;
        console.log(typeof x); // "boolean"
        console.log(typeof undeclaredVar); //
        "undefined"
      */
    </script>
  </head>
  <body>
  </body>
</html>
```



SCOPE - DOMENIU VIZIBILITATE

- domeniul de vizibilitate (*scope*) al variabilelor: zona din program în care sunt declarate variabilele
- scope global, scope funcție (local), scope bloc
- variabilele declarate de o funcție părinte sunt vizibile în descendenții ei

```
// scope global
function fA () {
  //scope A
  function fB () {
    // scope B
  }
}
```

SCOPE - DOMENIU VIZIBILITATE

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="demo.css">
    <script type="text/javascript">
      /*
        function calc(an){
          var varsta = 2024 - an;
          function print(){
            const str = nume + " are " +
varsta
              + " ani, născută în " +
an;
            if (varsta >= 7) {
              const oldCat = true;
            }
            console.log(str);
            //console.log(oldCat); // eroare
          }
          print();
        }
      */
    </script>
  </head>
</html>
```

>

SCOPE - HOISTING

- ne putem referi la variabilele declarate cu **var** oriunde în domeniul lor de vizibilitate, chiar și înainte de punctul în care sunt declarate
- înainte de a fi executat, codul JavaScript este "rearanjat" a.î. toate declarațiile de variabile (nu și operațiile de atribuire) sunt mutate (ridicate) la începutul zonei de vizibilitate

SCOPE - HOISTING

```
var x = 5; //globală
function hoist() { if (x == 5) { var x = 10; }
                  alert(x); }
hoist(); // va afișa undefined
alert(x); // va afișa 5
```

~

```
var x = 5; //globală
function hoist() { var x; // variabilă locală
                  if (x == 5) { // undefined
                      var x = 10; }
                  alert(x); }
hoist(); // va afișa undefined
alert(x); // va afișa 5
```

TIPUL NUMBER

- reprezentare binară pe 64 biți; întregi și float

```
var i = 4;  
var r = 42.2;
```

- operatorii aritmetici specifici: + - * / % ++ --
- conversie de tip automată:

```
x = "2" * 7; // 14  
y = "2" + 7; // "27"  
z = parseInt("2") + 7; // 9  
t = "2" * "7"; // 14  
u = 2 + 3 + "4"; // "54"  
z = "2" + 3 + 4; // "234"
```

TIPUL NUMBER

obiectul built-in **Math**

```
Math.PI //=> 3.14  
Math.pow(2,3) // => 8  
Math.round(4.7) // => 5  
Math.random() // între 0 și 1  
Math.sqrt(-1) // => NaN
```

TIPUL BOOLEAN

orice valoare poate fi convertită explicit folosind
obiectul predefinit Boolean

```
var nume = Boolean(valoare);  
true === Boolean("adevarat") // => true  
false === Boolean("") // => true
```

- operatori logici pentru tipuri primitive:
> < <= >= && || ! == === != !==
- putem folosi și alte tipuri de date în context logic:
 - pentru fals: 0, "", NaN, null, undefined
 - pentru adevărat: orice altă valoare

TIPUL STRING

șir de caractere scris între ' ' sau " "

```
var s = "Tigger Lilly";  
var t = 'Tigger Lilly';  
var pnume = s.slice (0, s.indexOf(" ")); // 'Tigger'  
var nume = s.slice(s.lastIndexOf(" ")+1,s.length); // 'Lilly'
```

proprietăți și metode:

```
length, charAt(), indexOf(), lastIndexOf(), replace(),  
split(), toLowerCase(), concat()
```

TIPUL STRING

- concatenare: "numărul" + "1", "numărul" + 1
- caractere speciale: \' \' " \n \t \v \b \\\
- accesarea unui caracter: s[pozitie], s.charAt(pozitie)

```
var x = "abcde";  
alert(x[0]);  
x[0] = 'v';  
alert(x[0]); // => a
```

TIPURILE UNDEFINED & NULL

- variabilele care nu au primit încă o valoare au tipul **undefined**

```
var x
x == undefined // true
typeof(x) // undefined
```

- **null**: lipsa unei valori (intenționată)

```
var x = null
typeof(x) // object !
null == undefined // true
null === undefined // false
```

TIPUL OBJECT

- un obiect este o colecție de perechi proprietate-valoare
- dacă valoarea este o funcție, atunci proprietatea se numește metodă
- ```
var ob = {prop1: val1, prop2: val2, ... , propn: valn};
```
- accesarea proprietăților:

```
ob.prop1; // val1
ob["prop1"]; // val1
```



# TIPUL OBJECT

```
var student = { nume: "Tigger",
 nota1: 9,
 nota2: 10,
 media: function() {
 return (this.nota1 + this.nota2)/2;
 }
 }
```

```
student.nume \\ "Tigger"
student.nota1 \\ 9
student.nota2 \\ 10
student.media() \\ 9.5
student.media \\ funcția
```

# TIPUL OBJECT

- toate datele din tipurile primitive sunt transmise prin valoare
- datele de tip object sunt transmise prin referință

```
var a = {nume: "Tigger"} // object
var b = a; // a și b referă aceeași zonă
b.nume = b.nume + " Lilly"; // se modifică și b și a
alert(a.nume); // "Tigger Lilly"
```

```
var s = "Tigger"; // string
var t = s; // t copiază valoarea lui s
t = t + " Lilly" ; // se modifică doar t
alert(s) // => "Tigger"
```

# CREAREA OBIECTELOR

- prin **object literal**
- proprietățile, metodele, împreună cu valorile lor sunt enumerate între acolade
- se creează un singur obiect

```
var cat = {nume:"Tigger", culoare:"neagră", vârstă:14}
```

# CREAREA OBIECTELOR

- cu ajutorul **obiectului generic**
- se apelează constructorul `new Object()` și se adaugă apoi proprietățile și metodele
- se creează un singur obiect

```
var cat = new Object();
cat.nume = "Tigger";
cat.culoare = "neagră";
cat.vârstă = 14;
```

# CREAREA OBIECTELOR

- cu ajutorul unui **constructor de obiecte**
- se definește o funcție `constructor(param)` care apoi va fi apelată cu `new constructor(param)` pentru fiecare obiect care va fi creat

```
function cat(n, c, v) {
 this.nume = n;
 this.culoare = c;
 this.vârstă = v;
}

var c1 = new cat("Tigger", "neagră", 14);
var c2 = new cat("Fluff", "albă", 2);
```

# BUILT-IN OBJECTS

proprietăți globale: returnează o valoare simplă

- Infinity
- NaN
- undefined

# BUILT-IN OBJECTS

metode globale: apelate global, nu pentru un obiect

- `isNaN()` - determină dacă valoarea este un număr invalid
- `parseInt()` - convertește un șir într-un întreg
- `parseFloat()` - convertește un șir într-un float
- `Number()` - convertește un obiect într-un număr
- `String()` - convertește un obiect într-un șir

# OBIECTE PREDEFINITE

- obiecte corespunzătoare tipurilor primitive  
String, Number, Boolean
- Array, Set, Map, Function, RegExp, Date
- se pot crea obiecte noi cu new



# OBIECTE PREDEFINITE

```
<!DOCTYPE html>
<html>
 <head>
 <link rel="stylesheet" href="demo.css">
 <script type="text/javascript">
 /*
 var y = new Number(123);
 alert(typeof(y)); // object

 var ob = new Object();
 ob.x = 1; ob.y = 2; // ob = {x:1, y:2}

 var d = new Date(2023,2,27);
 alert(d.getUTCDay()); // 0 (ziua din
 săptămână (0-6))
 */
 </script>
 </head>
 <body>
 </body>
</html>
```

>

# OBIECTE PREDEFINITE - ARRAY

```
var v = new Array();
v[0] = "a"; v[1] = "b";
```

```
var v = new Array("a", "b");
```

```
var v = ["hi", 2, [5, 7]];
v = [6, 4, 7, 3];
```

```
v.length; // 4
v.push(10); // => v = [6, 4, 7, 3, 10]
v.pop(); // => v = [6, 4, 7, 3]
v.shift(); // => v = [4, 7, 3]
v.unshift(10); // => v = [10, 4, 7, 3]
v.sort(); // => v = [3, 4, 6, 7]
```

# OBIECTE PREDEFINITE - ARRAY

```
var s = "azi este luni";
var a = s.split(" "); // a = ["azi", "este", "luni"]
a.reverse(); // a = ["luni", "este", "azi"]
var s = a.join('/'); // s = "joi/este/azi"
```

# FUNCȚII

```
function nume(arg1, arg2, ..., argn) {
 instrucțiuni;
 [return valoare;] // optional
}
```

o funcție JavaScript poate fi apelată cu un număr variabil de parametri

```
function suma(a,b) {
 return a+b;
}

suma(2,3); // 5
suma(); // NaN
suma(2); // NaN
suma(3,4,1,5,6,7) // 7
```

# FUNCȚII - ARGUMENTS

orice funcție poate accesa un obiect "arguments" (asemănător unui array) care conține valorile argumentelor cu care se apelează funcția

```
function fun() {
 return arguments.length;
}
```

```
fun(2, "sss", 5); // 3
```

# FUNȚII - ARGUMENTS

```
function func(a, b, c) {
 console.log(arguments[0]); // expected output: 1
 console.log(arguments[1]); // expected output: 2
 console.log(arguments[2]); // expected output: 3
}

func(1, 2, 3);
```

# FUNCȚII - ARGUMENTS

```
<!DOCTYPE html>
<html>
 <head>
 <link rel="stylesheet" href="demo.css">
 <script type="text/javascript">
 /* var n = prompt("gimme some number!");
 alert('suma este ' + suma(parseInt(n)));

 function suma() {
 var s = 0;
 for(var i=0; i<arguments.length; i++)
 s+=arguments[i];
 return s;
 } */
 </script>
 </head>
 <body>
 </body>
</html>
```

>

# FUNȚII ANONIME

## function expressions

```
function (arg1, ..., argn) {
 instrucțiuni;
}
```

```
const fun = function () {
 return arguments.length;
}
```

## arrow functions

```
var f = (x) => { x+1 }
```



# ARRAY - MAP()

```
array.map(function(currentValue, index, arr), thisValue)
```

- creează un nou array prin apelarea unei funcții (dată ca parametru) pentru fiecare element din array
- nu modifică array-ul inițial
- *index, arr, thisValue* sunt opționale

# ARRAY - MAP()

```
function f(x) {
 return x+1;
}
var array = [1,2,3,4];
var array1 = array.map(f);
console.log(array1); // [2,3,4,5]
```

```
function g(x) {
 return x * this.a;
}
var o1 = {a:2}, o2 = {a:3};
var array = [1,2,3,4];
var array1 = array.map(g,o1); // [2,4,6,8]
var array2 = array.map(g,o2); // [3,6,9,12]
```

# ARRAY - FOREACH()

```
array.forEach(function(currentValue, index, arr), thisValue)
```

- apelează o funcție dată ca parametru pentru fiecare element dintr-un array
- *index, arr, thisValue* sunt opționale

# ARRAY - FOREACH()

```
array.forEach(function(currentValue, index, arr), thisValue)
```

```
function f(x, i) {
 alert(i + " : " + x);
}
```

```
var array = ['luni', 'marți', 'miercuri', 'joi'];
array.forEach(f);
```

```
var d = [1, 2, 3, 4];
d.forEach(function(v, i, a) {a[i] = v + 1;});
console.log(d); // [2,3,4,5];
```

# ARRAY - FILTER()

```
array.filter(function(currentValue, index, arr), thisValue)
```

- creează un nou array cu elementele care verifică condiția implementată de funcția dată ca parametru
- nu modifică array-ul inițial
- *index, arr, thisValue* sunt opționale

# ARRAY - FILTER()

```
function check(word) {
 return word.length < 6;
}

var animals = ["cat", "tapir", "platypus", "red panda"];
var smol = animals.filter(check);
console.log(smol); // ["cat", "tapir"]
```

# ARRAY - REDUCE()

```
array.reduce(function(total, currentValue, currentIndex, arr),
```

- execută o funcție de reducere (dată ca parametru) pentru fiecare element al array-ului rezultând o singură valoare de ieșire
- nu modifică array-ul inițial
- *currentIndex, arr, initialValue* sunt opționale

# ARRAY - REDUCE()

```
function suma(total,val) {
 return total+val;
}

var array = [1,2,3,4];
var s = array.reduce(suma);
console.log(s); // 10;
```



# ARRAY

```
var cats = [{nume:"Tigger",ani:7},{nume:"Whiskers",ani:6},
 {nume:"Fluff",ani:3},{nume:"Kit",ani:2},
 {nume:"Tom",ani:6},{nume:"Berlitz",ani:4}];
var copy = cats.slice();
var oldCats = copy.filter((x) => (x.ani >= 6))
 .map((x) => (x.nume+" are "+x.ani+" ani"))
 .forEach((x) => {alert(x)});
```

**CLASE?**



