Seminarul 1 – Bazele JavaScript (elemente esențiale)

Acest prim segment al seminarului introduce fundamentele limbajului **JavaScript**, subliniind evoluțiile de la sintaxa tradițională ES5 la facilitățile moderne din ES6+[1]. Vom examina modul de declarare a variabilelor (vechiul `var` față de noile `let` și `const`), definirea funcțiilor (atât cu sintaxa clasică `function`, cât și folosind funcțiile „săgeată" introduse în ES6) și caracteristici precum parametrii cu valoare implicită și operatorii rest/spread. De asemenea, analizăm structurile de iterare (bucle `for`, `while`, `do…while`, `for…of`) și comparăm bucla modernă `for...of` cu metoda funcțională `Array.forEach()`. În final, vom vedea cum se pot citi argumente din linia de comandă în Node.js folosind `process.argv`, precum și o scurtă comparație conceptuală între platforma Node.js și framework-ul Express.js, alături de diferențele de utilizare între aplicațiile de tip CLI și cele web.

Declararea variabilelor în JavaScript: **var** vs **let** vs **const**

În JavaScript, există trei cuvinte cheie pentru declararea variabilelor: `var`, `let` și `const`, fiecare cu propriile caracteristici de *scop* (scope), comportament la *hoisting* și reguli de (re)atribuire[2]:

- **`var`** – Declară o variabilă cu *domeniu de vizibilitate* la nivel de funcție sau global, în funcție de locul unde este definită. Permite *redeclararea* și *reasignarea* variabilei în același scop (aceeași funcție sau context global)[3]. Variabilele `var` sunt *hoist*-ate (ridicate) la începutul funcției, inițializându-se implicit cu `undefined` dacă sunt accesate înainte de declarație[4].
- **`let`** – Introducerea în ES6 a cuvântului cheie `let` aduce *scopul la nivel de bloc* (block scope) pentru variabile[5]. O variabilă declarată cu `let` este vizibilă doar în interiorul blocului delimitat de `{ }` în care a fost definită (de exemplu, într-un `if` sau `for`)[5]. `let` permite *reasignarea* valorii, însă **nu permite redeclararea** unei variabile cu același nume în același bloc[6]. Spre deosebire de `var`, o variabilă `let` nu poate fi accesată înainte de linia unde este declarată – accesarea înainte de declarare ar duce la *ReferenceError* (așa-numita **„zonă moartă temporară"** / TDZ)[7].
- **`const`** – Tot cu scop de bloc, `const` definește o variabilă *constantă*, care **nu poate fi reatribuită** după inițializarea inițială[6]. Este obligatoriu să-i se atribuie o valoare inițială în momentul declarării (nu poate fi declarată fără valoare)[8]. Ca și `let`, o constantă `const` nu poate fi redeclarată în același bloc și nu poate fi accesată înainte de declarație (este și ea supusă TDZ)[4]. În cazul obiectelor sau array-urilor declarate cu `const`, constanta menține referința (nu poate fi înlocuită cu alt obiect), însă proprietățile interne ale obiectului pot fi modificate.

Recomandare: În codul modern, se preferă utilizarea lui `const` ori de câte ori este posibil (pentru valorile care nu trebuie reatribuite), și `let` dacă variabila va fi modificată, pe când ***var*** **ar trebui evitat** pentru a preveni efecte neașteptate datorate hoisting-ului și scopului neprevizibil[9][3].

Exemplu: Diferența de *scop* între `var` și `let` într-o structură de bloc:

```
function testVarLet() {
  if (true) {
    var a = 10;
    let b = 20;
  }
  console.log(a); // 10  (var este funcționalscoped, accesibil aici)
  console.log(b); // ReferenceError: b is not defined (let este block-scoped)
}
testVarLet();
```

```
const c = 5;
c = 6; // TypeError: Assignment to constant variable.
```

În exemplul de mai sus, variabila a declarată cu `var` este accesibilă în afara blocului `if`, pe când `b` declarată cu `let` *nu* există în afara acelui bloc. De asemenea, încercarea de a reatribui o constantă `c` produce o eroare de tip.

Funcții în JavaScript

Funcțiile reprezintă unități de bază ale codului JavaScript, permițând organizarea logicii în subrutine reutilizabile. Vom prezenta mai multe moduri de definire a funcțiilor și elemente asociate, de la sintaxa clasică la facilitățile moderne ES6.

Definirea funcțiilor: declarații clasice vs. funcții „săgeată”

În JavaScript ES5 (și anterior), funcțiile se defineau fie prin **declarații** cu cuvântul cheie `function`, fie ca **expresii de funcție** asignate unor variabile. ES6 a introdus o nouă sintaxă concisă: **funcțiile săgeată** (arrow functions), care elimină cuvântul `function` și folosesc simbolul `=>`.

Declarație clasică:

```
// Declarație de funcție clasică
function adună(x, y) {
  return x + y;
}
```

Expresie de funcție:

```
const adună2 = function(x, y) {
  return x + y;
};
```

Funcție săgeată (ES6+):

```
const adună3 = (x, y) => x + y;
```

În exemplul de mai sus, `adună`, `adună2` și `adună3` au comportament echivalent – toate întorc suma `x + y`. Funcția *săgeată* `adună3` este însă mai concisă: nu necesită sintaxa `function` și nici măcar `return` atunci când corpul funcției este o singură expresie (valoarea expresiei este returnată implicit). Observați de asemenea că la funcțiile săgeată cu un singur parametru se pot omite parantezele în jurul parametrului, iar pentru zero sau mai mulți parametri sunt necesare parantezele. Dacă corpul funcției săgeată are mai multe instrucțiuni, va trebui folosit blocul `{ }` și o instrucțiune explicită `return` când dorim să returnăm o valoare.

Funcțiile săgeată vin însă și cu **diferențe semantice deliberate** față de funcțiile obișnuite[10]:

- O funcție săgeată **nu are propriul context** `this` (nici propriul obiect `arguments`). Contextul `this` din interiorul ei este preluat lexical de la contextul extern (cel în care funcția este definită)[11]. Acest lucru le face utile în special pentru callback-uri, unde comportamentul clasic al lui `this` putea fi confuz.
- Funcțiile săgeată **nu pot fi folosite ca constructori** (nu se pot apela cu `new`); orice încercare de a folosi `new` pe o funcție săgeată va genera o eroare de tip[12]. Ele sunt menite doar ca funcții obișnuite, nu ca clase sau constructori.
- O funcție săgeată nu are acces la cuvântul cheie `yield` și nu poate fi folosită ca *generator*. Pentru generatori se utilizează în continuare sintaxa clasică `function*`.

În concluzie, funcțiile săgeată reprezintă un **înlocuitor compact** pentru funcțiile anonime clasice, oferind o sintaxă simplificată și un model de `this` mai predictibil (lexical)[13]. Totuși, pentru metode

ale obiectelor (care necesită propriul `this` legat la instanță) sau pentru constructori, trebuie folosite în continuare funcțiile tradiționale.

Exemplu: Diferență sintactică între funcție declarată și funcție săgeată, cu același rezultat:

```
function salut(nume) {
  return "Bună, " + nume + "!";
}
const salutArrow = (nume) => "Bună, " + nume + "!";


console.log(salut("Ana"));        // Bună, Ana!
console.log(salutArrow("Ana"));   // Bună, Ana!
```

Ambele versiuni afișează același mesaj; versiunea săgeată este mai concisă. Trebuie notat că, dacă am fi folosit `this` în interior, funcția săgeată **nu** ar crea un nou `this` propriu, ci l-ar folosi pe cel din contextul exterior.

Parametri cu valoare implicită (default) și parametri *rest/spread*

Parametri default: ES6 a introdus posibilitatea de a specifica **valori implicite pentru parametrii** unei funcții. Astfel, dacă la apelul funcției anumite argumente nu sunt furnizate (sau sunt `undefined`), vor fi folosite valorile default definite. Aceasta simplifică mult codul față de vechea abordare (în ES5) în care se verificau manual valorile `undefined` în corpul funcției[14][15].

Exemplu de definiție cu parametru implicit:

```
function salutare(nume = "Vizitator") {
  console.log(`Salut, ${nume}!`);
}
salutare("Ion");    // Salut, Ion!
salutare();         // Salut, Vizitator!
```

În exemplul de mai sus, funcția `salutare` are parametrul `nume` cu valoarea implicită `"Vizitator"`. La apelul `salutare("Ion")`, parametrul primește valoarea "Ion", suprascriind implicitul. La apelul fără argument, `nume` rămâne `"Vizitator"`, evitându-se astfel obținerea valorii `undefined` în interiorul funcției[16][17].

Parametrul *rest*: Sintaxa *rest* permite definirea unui parametru special precedat de `...` (trei puncte) care **colecționează toate argumentele suplimentare** transmise unei funcții într-un array. Cu alte cuvinte, putem defini funcții *variadice* care acceptă un număr nedefinit de argumente, fără a folosi obiectul vechi `arguments` (care, în plus, nu funcționează în funcțiile săgeată). Parametrul rest trebuie să fie ultimul în lista de parametri și va include tot ce "rămâne" neatribuit celorlalți parametri nominali[18].

Exemplu de funcție variadică cu rest:

```
function sumaTotală(...numere) {
  return numere.reduce((sum, nr) => sum + nr, 0);
}
console.log(sumaTotală(4, 7, 1));       // 12
console.log(sumaTotală(10, -5, 3, 1));  // 9
```

În `sumaTotală`, parametrul `...numere` va deveni un array ce conține toate argumentele cu care funcția este apelată (oricâte ar fi). Putem apoi să iterăm prin acest array (aici folosim `reduce` pentru a calcula suma). Dacă apelăm funcția fără argumente, array-ul va fi gol. Această abordare este mai curată și mai flexibilă decât manipularea manuală a obiectului `arguments` și este recomandată în codul modern[19].

Operatorul *spread*: Pe lângă parametrul rest (care *colectează* argumente într-un array), ES6 introduce și *operatorul spread* (`...`) care are efectul invers: **extinde (destructurează) un array sau obiect iterabil în elemente individuale**. Operatorul spread poate fi folosit la apelul unei funcții pentru a trece elementele unui array drept argumente separate, sau pentru a combina arrays/obiecte. Spre exemplu:

```
const valori = [4, 8, 2];
console.log( Math.max(...valori) ); // 8  (spread transmite 4,8,2 ca argumente
separate)
```

În exemplul de mai sus, în loc să apelăm `Math.max(4, 8, 2)`, putem obține același rezultat folosind `Math.max(...valori)`. Operatorul `...` a *împrăștiat* array-ul `valori` în argumente individuale pentru funcție[20][21]. De asemenea, putem folosi spread pentru a concatena rapid array-uri: de exemplu `let combinat = [...arr1, ...arr2]` îmbină două array-uri, sau pentru a copia obiecte: `let copie = { ...original }`.

Notă: Sintaxa `...` are deci dublu rol, în funcție de context: în **definirea** funcției, `...` denotă un parametru *rest* ce *adună* argumente într-un array; în **apelul** funcției sau în construirea de array/obiect, `...` acționează ca *spread* care *răspândește* elementele unui iterabil[21].

Recursivitatea funcțiilor

Recursivitatea este o tehnică de programare în care o funcție se apelează pe ea însăși, direct sau indirect, pentru a rezolva o problemă prin subdivizare în sub-probleme mai mici de același tip. O funcție recursivă trebuie să aibă o condiție de oprire (*casă de bază*) pentru a evita apelurile infinite. Recursivitatea este utilă în calcularea unor serii matematice sau parcurgerea unor structuri de date arborescente.

Exemplu clasic de funcție recursivă: calcularea **factorialului** unui număr *n*. Factorial *n!* se definește recursiv ca *n * (n-1)!*, cu baza 0! = 1.

```
function factorial(n) {
  if (n <= 1) {
    return 1;               // cazul de bază: factorial(0) = 1, factorial(1) = 1
  }
  return n * factorial(n - 1);  // apel recursiv pentru n > 1
}
console.log( factorial(5) );  // 120  (calculează 5*4*3*2*1)
```

În exemplul de mai sus, funcția `factorial` se apelează pe sine însăși cu o valoare mai mică (`n-1`) până ajunge la baza 1. Recursivitatea va fi întâlnită și în **Lab 1** pentru calculul numerelor Fibonacci. Deși recursivitatea poate oferi soluții elegante, trebuie folosită cu grijă: apelurile recursive excesive pot duce la stivă prea adâncă sau performanță scăzută, iar uneori o soluție iterativă este mai eficientă. În practica JavaScript modernă, pentru probleme intensive se preferă adesea soluții iterative sau folosind *built-in methods* ale array-urilor, dar recursivitatea rămâne un concept important de înțeles.

Structuri de iterare (bucles)

JavaScript oferă mai multe construcții pentru a repeta execuția unui bloc de cod, adică pentru a itera: bucla clasică cu indice (`for`), bucla cu condiție (`while`), varianta `do...while`, precum și construcții mai noi cum este `for...of` pentru iterarea colecțiilor. În plus, array-urile au metode funcționale precum `forEach` ce permit parcurgerea elementelor. Vom trece în revistă aceste mecanisme, evidențiind când și cum se folosesc.

Bucle **for**, **while** și **do...while**

- **`for` (clasic):** bucla `for` este utilă atunci când cunoaștem numărul de iterații sau dorim să iterăm o secvență incrementând/decrementând un contor. Sintaxa generală: `for (initializare; condiție; expresie post-iterație) { /* cod */ }`. Exemplu:

```
for (let i = 0; i < 5; i++) {
  console.log("Iterația nr. " + i);
}
// Va afișa i de la 0 la 4
```

  În exemplul de mai sus, bucla pornește cu `i=0` și rulează atâta timp cât `i < 5`, incrementând `i` la fiecare iterație.

- **`while`:** bucla `while` repetă un bloc de cod *cât timp o anumită condiție este adevărată*. Este potrivită când numărul de iterații nu este cunoscut din start, dar avem o condiție de continuare. Exemplu:

```
let j = 5;
while (j > 0) {
  console.log(j);
  j--;
}
// Afișează 5,4,3,2,1
```

  Aici, codul din buclă se execută atâta timp cât variabila `j` este mai mare decât 0, fiind decrementată la fiecare pas. Dacă condiția inițială este falsă, bucla `while` nu execută nimic (posibil 0 iterații).

- **`do...while`:** această buclă este similară cu `while`, dar cu diferența că **condiția este evaluată la** sfârșitul **buclei**, nu la început. Astfel, blocul de cod din `do { ... } while(cond)` se execută *cel puțin o dată*, apoi continuă cât timp condiția `cond` este adevărată. Exemplu:

```
let k = 0;
do {
  console.log("K este " + k);
  k++;
} while (k < 3);
// Afișează "K este 0", "K este 1", "K este 2"
```

  Chiar dacă condiția ar fi fost falsă de la început, bucla `do...while` execută totuși o iterție înainte de verificare. Acest tip de buclă e util când trebuie să executăm ceva **cel puțin o dată**, verificând abia apoi dacă trebuie să continuăm.

În alegerea între aceste bucle: `for` este adesea preferat pentru iterații cu indice cunoscut sau pentru parcurgerea array-urilor folosind indicele; `while` și `do...while` sunt utile pentru iterații bazate pe condiții mai complexe sau când nu știm câte iterații vor fi (ex: citirea dintr-un stream de date până la epuizare).

Iterarea colecțiilor: for..of vs Array.forEach()

Pentru parcurgerea elementelor unui array (sau a altor colecții iterabile, precum seturi, liste de noduri DOM etc.), ES6 a introdus bucla **`for...of`**, care simplifică mult sintaxa față de bucla `for` clasică. De asemenea, array-urile oferă metoda **`forEach()`**, care permite aplicarea unei funcții pe rând asupra fiecărui element. Vom compara pe scurt aceste abordări:

- **`for...of`:** Permite iterarea *directă a valorilor* dintr-o colecție iterabilă (array, string, Map, Set etc.). Sintaxa: `for (const element of colecție) { ... }`. La fiecare iterație, variabila specificată (`element`) ia pe rând valoarea fiecărui element din colecție. Exemplu:

```
const litere = ['a', 'b', 'c'];
for (const lit of litere) {
  console.log(lit);
}
// Afișează: a, b, c
```

În acest exemplu, `for...of` parcurge array-ul `litere` și atribuie fiecărei litere pe rând variabilei `lit`, pe care o afișăm. Bucla se termină automat când ajunge la sfârșitul colecției. Nu avem nevoie să gestionăm indici manual sau să verificăm condiții de terminare explicit – bucla *știe* când colecția s-a terminat.

- **`array.forEach(...)`:** Este o metodă a obiectului Array care primește o funcție callback și o apelează pentru fiecare element din array. Exemplu:

```
const litere = ['a', 'b', 'c'];
litere.forEach((element, index) => {
  console.log(index + ": " + element);
});
// Afișează: "0: a", "1: b", "2: c"
```

În exemplul de mai sus, funcția callback dată lui `forEach` primește automat parametri: valoarea elementului curent (`element`) și indexul acestuia (`index`). Astfel, putem folosi aceste valori în interiorul callback-ului. Spre deosebire de `for...of`, `forEach` *nu* necesită declarația unei variabile de iterare separate, și nici nu expune mecanismul iterației – totul este încapsulat în apelul metodei.

Diferențe cheie între `for...of` și `forEach()`:

- `for...of` este o structură de limbaj, iar `forEach` o metodă a obiectului Array. Ca urmare, `for...of` poate itera **orice** obiect iterabil (de ex. și un `Set` sau un `Map`, chiar și un `String` caracter cu caracter), pe când `forEach` se aplică direct doar pe Array (și pe structuri array-like care implementează manual o astfel de metodă).
- În interiorul unei bucle `for...of` putem utiliza declarații de control precum `break` sau `continue` pentru a opri iterarea anticipat sau a trece la următoarea iterație. În schimb, cu `forEach` **nu putem întrerupe** iterarea la mijloc – nu există suport pentru `break/continue` în callback (o soluție ocolită ar fi aruncarea unei excepții pentru a ieși din buclă, dar nu este recomandată). Dacă aveți nevoie să puteți ieși din loop pe baza unei condiții, `for...of` sau bucla `for` clasică sunt opțiuni mai potrivite.
- Din perspectiva conceptului de programare, `forEach` încurajează un stil *funcțional*: îi transmitem ce să facă pentru fiecare element, fără a ne ocupa manual de indexare. În schimb, `for...of` este mai aproape de stilul imperativ, deși mai concis decât un `for` clasic.
- În ceea ce privește **performanța**, diferențele sunt minime în majoritatea cazurilor practice. Bucla `for...of` este foarte eficientă și, de fapt, aduce mai puține *edge cases* comparativ cu o buclă tradițională sau `for...in` (care *nu* ar trebui folosit pentru array-uri). `forEach` are un mic overhead prin apelul repetat al funcției callback, dar în cod uzual diferența nu este semnificativă[22]. Alegerea se face mai mult pe criterii de lizibilitate și necesități de control al fluxului.

În general, **`for...of` este considerat o modalitate robustă și simplă de a itera un array** în JavaScript modern – codul este mai concis decât un `for` convențional și evită multe cazuri speciale pe care `for...in` sau chiar `forEach` le pot avea[23]. Totodată, `forEach()` rămâne util pentru concizie, mai ales când nu avem nevoie să întrerupem bucla și vrem să aplicăm o funcție fiecărui element (de exemplu, pentru efecte secundare sau operații pe fiecare element). Este important de reținut că, dacă avem nevoie să așteptăm operații asincrone în timpul iterării, `for...of` permite folosirea `await` în interiorul său, pe când `forEach` **nu** suportă direct `await` în callback (ceea ce poate duce la comportamente neașteptate)[24][25].

Exemplu comparativ:

```
const arr = ['a', 'b', 'c'];

// Iterare cu for...of (afișează doar valorile)
for (const valoare of arr) {
  console.log(valoare);
}
// Output: a, b, c

// Iterare cu forEach (afișează index și valoare)
arr.forEach((valoare, index) => {
  console.log(index + " -> " + valoare);
});
// Output: 0 -> a, 1 -> b, 2 -> c
```

În primul caz am parcurs array-ul cu `for...of` obținând direct valoarea fiecărui element. În al doilea caz, am folosit `forEach` cu o funcție care primește atât valoarea cât și indexul, pe care le-am afișat. Ambele metode parcurg întreaga listă. Dacă am dori însă să ne oprim la întâlnirea unei anumite valori, `for...of` ne-ar permite să folosim `break`, pe când cu `forEach` ar fi necesare artificii.

Citirea argumentelor din linia de comandă în Node.js (CLI)

Una dintre facilitățile oferite de **Node.js** (mediul de rulare JavaScript pe server sau consolă) este accesul la argumentele de linia de comandă. Când rulăm un script Node din terminal, putem transmite argumente adiționale, de exemplu: `node script.js arg1 arg2`. În interiorul scriptului, aceste argumente sunt expuse prin *proprietatea* globală `process.argv`.

`process.argv` este un array care conține argumentele cu care a fost lansat procesul Node[26]. Convenția este următoarea:

- `process.argv[0]` este calea către executabilul Node (de exemplu, `/usr/bin/node` sau similar).
- `process.argv[1]` este calea completă a scriptului JavaScript în execuție.
- `process.argv[2]`, `process.argv[3]`, ... reprezintă *argumentele efective* furnizate de utilizator în linia de comandă, în ordinea în care au fost scrise[26].

Astfel, lungimea lui `process.argv` va fi de **2 + numărul de argumente transmise** scriptului.

Exemplu: Să presupunem un script `procesare.js` care afișează lista argumentelor:

```
// procesare.js
process.argv.forEach((val, index) => {
  console.log(`${index}: ${val}`);
});
```

Dacă rulăm în terminal comanda:

```
$ node procesare.js unu doi=trei 4
```

Output-ul generat va fi:

```
0: /usr/local/bin/node      (execPath Node.js)
1: /cale/catre/procesare.js (scriptul rulat)
2: unu                      (primul argument)
3: doi=trei                 (al doilea argument)
4: 4                        (al treilea argument)
```

Exact acest comportament este descris și în documentația Node.js[26][27]. După cum vedem, *argumentele utile* pentru logica aplicației sunt cele de la index 2 încolo. De aceea, se folosește adesea `const args = process.argv.slice(2)` pentru a obține direct un array cu argumentele relevante (ignorând primele două poziții rezervate).

Argumentele sunt furnizate ca șiruri de caractere, deci dacă dorim să interpretăm un argument ca număr (de exemplu, pentru un script care primește un număr), trebuie să-l convertim manual (folosind parseInt, Number etc.). De asemenea, pentru argumente mai complexe (ex. *flags* tip `--option=value`), adesea se apelează la biblioteci de parsing (yargs, commander) – dar în esență toate pornesc de la `process.argv`.

Exemplu practic: În **Lab 1** vom folosi `process.argv` pentru a citi un număr introdus de utilizator și a calcula Fibonacci. Mai jos este un fragment demonstrativ despre cum extragem argumentele într-un script Node CLI:

```
// Exemplu simplu: citirea unui număr din argumente și afișarea pătratului
acestuia
const args = process.argv.slice(2);
if (args.length < 1) {
  console.log("Utilizare: node patrat.js <numar>");
  process.exit(1);
}
const nr = Number(args[0]);
console.log(`Pătratul lui ${nr} este ${nr * nr}`);
```

Dacă salvăm acest cod în `patrat.js` și rulăm `node patrat.js 5`, vom obține output-ul `Pătratul lui 5 este 25`. Am folosit `process.argv.slice(2)` pentru a obține argumentele user și apoi am convertit primul argument la număr cu `Number()`.

Node.js vs Express.js – O scurtă comparație

Node.js este un mediu de rulare (runtime) pentru JavaScript construit peste motorul V8 de la Google. Node permite executarea de cod JavaScript în afara unui browser, de obicei pe partea de server sau în mediul de linie de comandă. Vine cu un set de module standard (FS pentru sistem de fișiere, HTTP pentru servere web de bază etc.) și oferă posibilitatea de a gestiona operații de I/O în mod asincron, făcându-l ideal pentru aplicații server *scalabile*.

Express.js, pe de altă parte, este un *framework web* minimalist pentru Node.js. Practic, Express este o bibliotecă (un modul NPM) construită peste Node, care facilitează dezvoltarea aplicațiilor web (API-uri, site-uri web, servicii REST). În timp ce Node.js singur ne pune la dispoziție primitive low-level pentru a crea un server HTTP, a parsa manual cereri și a trimite răspunsuri, Express adaugă un strat de abstracție de nivel mai înalt, oferind **instrumente out-of-the-box pentru web**[28].

Ce aduce Express.js în plus față de Node pur? Iată câteva aspecte cheie:

- **Routing ușor al URL-urilor:** Express permite definirea de rute (adresă și metodă HTTP) și asocierea lor cu funcții *handler*. De exemplu, putem scrie `app.get("/utilizatori", funcție)` pentru a gestiona cereri GET către ruta `/utilizatori`. Fără Express, în Node ar trebui să inspectăm manual URL-urile și să scriem mult cod pentru rutare.
- **Suport pentru șabloane (views) și HTML dinamic:** Express se integrează cu motoare de template (pug, EJS, handlebars etc.) permițând generarea dinamică de pagini HTML pe server, cu efort redus.
- **Middleware:** Arhitectura Express se bazează pe middleware – funcții care interceptează cererile și pot executa diverse operații (autentificare, logare, parsare de body, etc.) înainte să ajungă la handler-ul final. Acest mod modular de a construi pipeline-ul de cereri/răspunsuri simplifică mult adăugarea de funcționalități cross-cutting.

- **Parsing-ul cererilor HTTP:** Express vine cu middleware-uri (sau suport prin biblioteci) pentru a parsa automat corpul cererilor (ex. date de formular JSON sau URL-encoded), lucru care Node simplu nu îl face implicit. De exemplu, pentru `POST` cu formular, Express poate pune direct datele în `req.body`.
- **Gestionarea sesiunilor și cookie-urilor:** Deși nu inclus direct în nucleul Express, există middleware-uri oficiale care se ocupă de sesiuni, cookies, autentificare, lucru care în Node pur ar necesita mult cod manual sau biblioteci disparate.
- **Servirea de fișiere statice:** Printr-o simplă configurație, Express poate servi pagini statice, fișiere CSS, imagini dintr-un director dedicat (`express.static` middleware), scutind efortul de a implementa manual streaming-ul acestor resurse.
- **Ecosistem bogat de module:** Fiind atât de popular, Express are mii de module third-party (pe NPM) care se integrează ușor (ex. pentru validarea input-ului, protecție CSRF, upload de fișiere, etc.), permițând obținerea rapidă a unei configurații de server web complete[29].

Pe scurt, **Express este pentru Node ceea ce un framework web este pentru un limbaj de programare**: îți oferă un schelet și componente gata făcute pentru a construi aplicații web robuste, fără să reinventezi roata pentru fiecare aplicație. O comparație celebră spune că Express pentru Node este similar cu ceea ce este *Sinatra* pentru Ruby[30] – un micro-framework flexibil care se ocupă de routing, views, sesiuni ș.a. Node singur *poate* face orice face Express (fiind suficient de low-level), dar ar necesita mult mai mult cod și cunoștințe fine de protocol HTTP. De exemplu, **Node.js „nu știe" conceptul de routing sau de parametri de query, nu are noțiunea de `req.body` pre-parsat**, acestea trebuind implementate de la zero; Express *"face web"*, adică include toate aceste mecanisme comune aplicațiilor web, pe când Node oferă doar bazele (*I/O, http server barebone*)[31].

Un alt mod de a privi lucrurile: Node este motorul și baza, Express este caroseria și direcția care te ajută să construiești mai ușor un „vehicul" web. Dacă vrei să creezi rapid un API REST sau un site, vei alege aproape sigur Express (ori alt framework similar) în locul Node pur, deoarece câștigi timp și simplitate în dezvoltare. Desigur, există și alte framework-uri peste Node (Koa, Hapi, Fastify etc.), dar Express rămâne cel mai cunoscut și folosit datorită simplității și flexibilității sale.

Referință: Documentația oficială subliniază că Express este un *web framework* pentru Node.js construit peste middleware-ul Connect, permițând crearea de site-uri și aplicații web în mod foarte facil, cu suport pentru routing, view-uri, sesiuni și altele[28]. În contrast, Node.js singur este considerat mult mai *low-level*, lăsând la latitudinea programatorului implementarea acestor facilități sau folosirea de module separate[31].

Utilizarea în linie de comandă vs. utilizarea web (CLI vs Web)

Aceeași aplicație (logică) poate fi expusă utilizatorilor fie printr-o interfață de linie de comandă (CLI), fie printr-o interfață web (de obicei în browser, posibil cu un server backend). Fiecare abordare are avantajele și contextul ei de utilizare:

- **Aplicații de tip CLI (Command Line Interface):** Rulează în consolă/terminal și interacționează cu utilizatorul prin text (comenzi introduse de utilizator, mesaje afișate de program).

  Avantaje:

  - *Simplitate în implementare – nu necesită dezvoltarea unei interfețe grafice, doar parsează input text și produce output text.*
  - *Pot fi ușor automatizate – pentru sarcini repetitive, scripturile CLI pot fi incluse în shell scripts, pipeline-uri etc.*
  - *Consum redus de resurse – rulează local, fără nevoie de browser, potrivit pentru unelte pentru programatori sau administratori (ex: utilitare, procesări de fișiere, migrări de date).*

- *Mediu unificat – codul care rulează e tot JavaScript (în Node), nu există diferența client/server (totul e local).*

Dezavantaje:

- ***Accesibilitate scăzută pentru non-tehnici:*** *utilizatorul trebuie să fie confortabil cu terminalul și să știe să lanseze comenzi.*
- *Interacțiune limitată – doar text brut, fără elemente vizuale sau controale interactive (în afara cazului în care se folosesc biblioteci speciale pentru CLI interactiv).*
- *Distribuție limitată – de obicei folosit intern sau de un număr restrâns de utilizatori (nu „expui" un program CLI pentru publicul larg în același mod ca o aplicație web).*

- **Aplicații Web (interfață web):** Au de obicei o componentă front-end (HTML/CSS/JS rulând în browser) și opțional o componentă back-end (server, adesea Node.js/Express, care furnizează date sau logica).

Avantaje:

- ***Ușurință în utilizare pentru publicul larg:*** *oricine știe să folosească un browser poate accesa aplicația, interfața fiind grafică și intuitivă. Nu cere cunoștințe tehnice deosebite.*
- ***Interactivitate bogată:*** *putem avea formulare, butoane, grafice, animații, feedback imediat către utilizator, elemente multimedia etc., ceea ce îmbunătățește experiența.*
- ***Acces de oriunde:*** *prin natura web, aplicația poate fi accesată de pe orice dispozitiv cu internet, fără instalare (doar cu un URL).*
- ***Integrare cu servicii web:*** *ușor de conectat la baze de date, servicii externe, autentificare, și de deservit mai mulți utilizatori simultan (scalabilitate prin server).*

Dezavantaje:

- ***Complexitate mai mare în dezvoltare:*** *trebuie gestionat atât codul de client (browser) cât și codul de server, plus comunicarea dintre ele (HTTP, API-uri). Debugging-ul poate fi mai complicat din cauza naturii distribuite (front-end vs back-end).*
- ***Necesită infrastructură de server:*** *pentru o aplicație web ai nevoie de un server (sau serviciu cloud) pe care să ruleze back-end-ul, ceea ce implică costuri și administrare. Chiar și aplicațiile pur front-end (SPA) trebuie găzduite undeva.*
- ***Performanță percepută poate varia:*** *depinde de conexiune internet, de puterea dispozitivului client etc., pe când un program CLI local rulează în general foarte rapid dacă are resurse.*

Când să alegi CLI vs Web? Depinde de scopul aplicației și de publicul țintă. Dacă realizezi un utilitar pentru dezvoltatori sau administratori de sistem (ex: un script de deployment, un generator de proiect, un procesator de date), o interfață CLI are sens – utilizatorii sunt tehnici și vor aprecia un tool scriptabil. În schimb, dacă aplicația este menită pentru utilizatori obișnuiți sau are nevoie de o interfață prietenoasă (ex: un site de comerț, o aplicație de notițe, un joc online), atunci o soluție web este preferabilă. Uneori, aceeași logică poate fi oferită în ambele moduri: de exemplu, un motor de calcul (cum e cel de Fibonacci din laborator) poate fi expus ca unealtă CLI pentru programatori, dar și ca pagină web demonstrativă pentru uz general.

Mediul de rulare diferit: Merită menționat că, în modul CLI, codul JavaScript rulează în Node.js (deci are acces la sistemul de fișiere, la `process.argv`, la modulul `fs`, etc., dar nu are DOM sau elemente de interfață). În modul Web (frontend), codul JS rulează în browser (deci are acces la DOM, poate manipula HTML-ul, dar *nu* are acces direct la sistemul de fișiere al clientului sau la `process` din Node). Dacă folosim Node.js și pentru server, atunci avem și acolo un mediu Node *fără DOM*, iar comunicarea între browser și server se face prin HTTP (AJAX/fetch API, WebSocket sau alte protocoale). Cu alte cuvinte, **JavaScript-ul de front-end și cel de back-end, deși folosesc același limbaj, rulează în medii diferite** și au API-uri disponibile diferite.

În concluzie, **CLI vs Web** nu sunt neapărat în opoziție, ci două medii diferite unde putem rula logica noastră JavaScript. Vom vedea în laboratoarele următoare exemple concrete ale ambelor abordări.

LABORATOARE

Pentru a consolida cunoștințele, vom realiza două aplicații simple atât în variantă CLI (Node.js) cât și în variantă Web (HTML + JS în browser). Scopul este să exersăm citirea input-ului, utilizarea funcțiilor, a structurilor de control și să evidențiem diferențele de mediu de rulare.

Laborator 1: Calculator Fibonacci – versiune CLI și versiune Web

Descriere: Calculați al n-ulea număr din șirul Fibonacci, unde șirul Fibonacci este definit astfel: $F(0)=0$, $F(1)=1$, iar $F(n)=F(n-1)+F(n-2)$ pentru $n \geq 2$. Aplicația va oferi o soluție în linie de comandă (CLI) și una web (pagina HTML cu JavaScript).

1. **Versiunea CLI (Node.js):**

Creați un fișier JavaScript (ex: `fibonacci-cli.js`) care să citească din `process.argv` un număr întreg *N* (indexul din șirul Fibonacci) și să afișeze în consolă valoarea F(N). Programul ar trebui să verifice că argumentul există și este valid (număr întreg pozitiv, reasonably small pentru a putea fi calculat). Puteți implementa funcția de calcul fie recursiv (direct pe baza definiției), fie iterativ (eficient, cu un loop). Exemplu de rulare a programului și output așteptat:

```
$ node fibonacci-cli.js 6
Fib(6) = 8
```

(În acest exemplu, scriptul a calculat al 6-lea număr Fibonacci, care este 8.)

O posibilă implementare (recursivă) a funcției Fibonacci în Node CLI:

```
function fib(n) {
  if (n < 2) return n;
  return fib(n - 1) + fib(n - 2);
}
const args = process.argv.slice(2);
const n = parseInt(args[0]);              // citim N din argumentele CLI
if (isNaN(n)) {
  console.error("Te rog introdu un număr valid.");
  process.exit(1);
}
console.log(`Fib(${n}) = ${fib(n)}`);
```

2. **Versiunea Web (HTML + JS):**

Realizați o pagină HTML simplă (autonomă, care nu necesită server special – poate fi deschisă direct în browser) ce conține un mecanism prin care utilizatorul să introducă un număr *N* și să obțină calculul lui Fibonacci. Se poate folosi fie un prompt modal simplu, fie elemente de

formular pe pagină. De exemplu, pagina poate avea un input text sau number pentru N și un buton "Calculează", rezultatul fiind afișat într-un element <p> sau în consolă. Logica de calcul va fi scrisă în JavaScript (același algoritm ca în versiunea CLI, dar adaptat la browser).

Un posibil exemplu minimal de pagină (puteți rula codul în **Firefox DevTools** sau orice browser modern pentru testare):

```
<!DOCTYPE html>
<html lang="ro">
<head>
  <meta charset="UTF-8" />
  <title>Fibonacci Calculator</title>
</head>
<body>
  <h3>Calculator Fibonacci</h3>
  <label for="n">Introdu un număr:</label>
  <input id="n" type="number" min="0" />
  <button onclick="calcFib()">Calculează</button>
  <p id="rezultat"></p>

  <script>
    function fib(n) {
      if (n < 2) return n;
      return fib(n - 1) + fib(n - 2);
    }
    function calcFib() {
      const nValue = document.getElementById('n').value;
      const n = parseInt(nValue);
      if (isNaN(n)) {
        alert("Te rog introdu un număr valid.");
        return;
      }
      const result = fib(n);
      document.getElementById('rezultat').textContent =
        `Fib(${n}) = ${result}`;
      console.log(`Fib(${n}) = ${result}`);
    }
  </script>
</body>
</html>
```

Salvați acest cod într-un fișier HTML (ex: `fibonacci.html`) și deschideți-l cu Firefox sau Chrome. Introduceți un număr în câmp și apăsați "Calculează" – rezultatul va apărea pe pagină și totodată este trimis către consola browserului (puteți verifica în DevTools console). Observați că aceeași funcție `fib(n)` a fost folosită, dar mecanismul de input/output diferă față de versiunea CLI: aici folosim DOM (document.getElementById etc.) și interacțiune grafică.

Notă: Algoritmul recursiv pentru Fibonacci are o complexitate exponențială, deci pentru valori mari ale lui N (peste ~40) devine lent. Scopul însă este exersarea recursivității; opțional, puteți implementa iterativ pentru eficiență. De asemenea, în practică am adăuga validări mai solide (ex. N să nu fie negativ, să nu fie prea mare etc.).

Laborator 2: Frequency Counter – versiune CLI și versiune Web

Descriere: Realizați o aplicație care să citească un șir de caractere de la utilizator și să calculeze frecvența de apariție a fiecărui caracter (sau fiecărei litere) din text. Cu alte cuvinte, un "numărător de frecvență" care, dat un input de forma "abbc", produce de exemplu rezultatul: a:1, b:2, c:1. Vom implementa acest utilitar atât ca script CLI cât și ca pagină web.

1. **Versiunea CLI (Node.js):**

Creați un script Node (ex: `freq-cli.js`) care preia **toate argumentele text** transmise (începând cu `process.argv[2]`) și le concatenează într-un singur string (sau, simplu, puteți presupune că textul de analizat e trecut ca un singur argument între ghilimele dacă conține spații). Apoi, parcurgeți textul și contabilizați frecvența fiecărui caracter. La final, afișați rezultatele în consolă, fiecare caracter urmat de numărul de apariții. Exemplu de rulare:

```
$ node freq-cli.js "abracadabra"
a: 5
b: 2
r: 2
c: 1
d: 1
```

(Exemplu: pentru inputul "abracadabra", se numără fiecare literă.)

O posibilă implementare simplă:

```
const args = process.argv.slice(2);
const text = args.join(" ");           // combină argumentele într-un
singur text
if (!text) {
  console.error("Folosește: node freq-cli.js \"<text>\"");
  process.exit(1);
}
const freq = {};
for (const char of text) {
  freq[char] = (freq[char] || 0) + 1;
}
for (const [char, count] of Object.entries(freq)) {
  console.log(`${char}: ${count}`);
}
```

În acest cod, folosim un obiect JavaScript simplu `freq` ca dicționar de frecvențe: parcurgem fiecare caracter `char` din text și incrementăm contorul corespunzător (`freq[char]`). La final, iterăm prin `Object.entries(freq)` pentru a afișa perechile caracter: număr. (Notă: `for...of` ne ajută să parcurgem direct caracterele unui string, deoarece un string este iterabil secvențial peste caractere.)

2. **Versiunea Web (HTML + JS):**

Pregătiți o pagină HTML (ex: `frequency.html`) care conține o zonă de text (ex. `<textarea>`) în care utilizatorul poate introduce un paragraf sau orice text, și un buton "Calculează frecvența". La apăsarea butonului, un script JavaScript va prelua textul din textarea și va calcula frecvența fiecărui caracter (sau, dacă doriți, poate ignora spațiile și semnele de punctuație, concentrându-se doar pe litere). Rezultatul poate fi afișat sub formular, de exemplu într-un `<div>` sau `<pre>` pentru formatare fixă.

Schiță posibilă de implementare:

```
<!DOCTYPE html>
<html lang="ro">
<head>
  <meta charset="UTF-8" />
  <title>Frequency Counter</title>
</head>
```

```
<body>
  <h3>Contor de frecvență caractere</h3>
  <textarea id="inputText" rows="4" cols="50" placeholder="Introdu textul
aici"></textarea><br/>
  <button onclick="calcFreq()">Calculează</button>
  <pre id="output"></pre>

  <script>
    function calcFreq() {
      const text = document.getElementById('inputText').value;
      const freq = {};
      for (let ch of text) {
        freq[ch] = (freq[ch] || 0) + 1;
      }
      let rezultat = "";
      for (let [ch, count] of Object.entries(freq)) {
        rezultat += `${ch}: ${count}\n`;
      }
      document.getElementById('output').textContent = rezultat;
      console.log("Rezultat frecvențe:\n" + rezultat);
    }
  </script>
</body>
</html>
```

Această pagină ia conținutul din `<textarea>` și, la apelul lui `calcFreq()`, construiește un obiect de frecvență similar cu versiunea CLI, apoi afișează rezultatele într-un element `<pre>` (pentru a păstra formatarea pe linii separate). Am folosit `textContent` pentru a insera textul, astfel încât caracterele precum newline `\n` să fie respectate (alternativ, am fi putut construi noduri DOM separate pentru fiecare rezultat). De asemenea, se loghează rezultatul în consola browserului pentru verificare.

Testare: Deschideți pagina HTML în Firefox, introduceți un text (de exemplu "Hello World!") în zona de text și apăsați Calculează. Ar trebui să vedeți sub buton ceva de genul:

```
H: 1
e: 1
l: 3
o: 2
 : 1
W: 1
r: 1
d: 1
!: 1
```

(acest exemplu include și spațiul ca și caracter numărat – puteți alege să excludeți spațiile dacă doriți, filtrând `if (ch !== ' ')` înainte de numărare).

Observați din nou diferența de mediu: în versiunea web folosim `document.getElementById` și manipulăm elemente HTML, în timp ce în versiunea Node CLI citim din `process.argv` și scriem în stdout. Logica de numărare în sine este practic aceeași în ambele cazuri, ceea ce ilustrează portabilitatea algoritmilor între medii, chiar dacă input/output-ul diferă.

Concluzie: Acest prim segment al seminarului a acoperit conceptele esențiale legate de sintaxa de bază JavaScript (declarații de variabile, funcții, iterație) și modul de rulare atât în mediu Node (CLI) cât și în mediu browser (Web), precum și rolul unui framework

ca Express.js în facilitarea dezvoltării web pe platforma Node. În laborator, veți pune în practică aceste cunoștințe, consolidându-le prin exemple concrete. Fie că scriem scripturi de linie de comandă sau pagini web, **fundamentele JavaScript rămân aceleași**, iar stăpânirea lor constituie baza pentru a progresa către dezvoltarea de aplicații web complexe.

BIBLIOGRAFIE

[1] What is ES6 in JavaScript? A Comprehensive Guide to ECMAScript 2015

https://talent500.com/blog/what-is-es6-javascript-guide/

[2] [3] [4] [5] [6] [8] Difference between var, let and const keywords in JavaScript - GeeksforGeeks

https://www.geeksforgeeks.org/javascript/difference-between-var-let-and-const-keywords-in-javascript/

[7] [9] let - JavaScript | MDN

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let

[10] [11] [12] [13] Arrow function expressions - JavaScript | MDN

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

[14] [15] [16] [17] Default parameters - JavaScript | MDN

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Default_parameters

[18] [20] [21] Rest parameters and spread syntax

https://javascript.info/rest-parameters-spread

[19] The arguments object - JavaScript | MDN - Mozilla

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments

[22] JS Array.foreach vs for-loops, pros and cons, which do you use and ...

https://www.reddit.com/r/webdev/comments/sxs9qd/js_arrayforeach_vs_forloops_pros_and_cons_which/

[23] [24] [25] For vs forEach() vs for/in vs for/of in JavaScript | www.thecodebarbarian.com

https://thecodebarbarian.com/for-vs-for-each-vs-for-in-vs-for-of-in-javascript.html

[26] [27] Process | Node.js v24.9.0 Documentation

https://nodejs.org/api/process.html

[28] [29] [30] [31] node.js - Confused with all the Node JS frameworks/libraries etc. around - Stack Overflow

https://stackoverflow.com/questions/7116332/confused-with-all-the-node-js-frameworks-libraries-etc-around