SEMINAR 4 – Obiecte și programare asincronă în Node.js

OBIECTIVE (*de învățare*)

Seminarul de față își propune să consolideze cunoștințele despre **programarea orientată pe obiecte (OOP) în JavaScript** și să introducă noțiuni de **programare asincronă în Node.js**. Vom parcurge exemple de cod care ilustrează:

- Crearea și **moștenirea** claselor în JavaScript, **prototipurile** și modul în care putem adăuga metode în timpul execuției.
- Utilizarea **compoziției** (obiecte conținând alte obiecte) și a **metodelor comune** în OOP (constructori, metode de instanță, metode prototip).
- Noțiuni de **asincronism**: mecanismul de evenimente în Node.js, folosirea **streamurilor** pentru I/O, **callback-uri**, **Promise**-uri și `async/await` pentru a orchestra operații care nu se execută instantaneu.
- Diferența dintre **codul sincron și cel asincron** în Node.js, ilustrată prin exemple de citire a fișierelor în mod blocant vs. neblocant.
- Tehnica de **memoizare** (cache) a rezultatelor unei funcții recursive costisitoare, ca exemplu de optimizare folosind structuri de date și concepte OOP (încapsulare prin închiderea unei funcții în altă funcție).

La finalul lecției veți găsi exerciții practice și o recapitulare a conceptelor învățate.

1. POO în JavaScript

În JavaScript, programarea orientată pe obiecte se poate realiza atât prin sistemul de **prototipuri**, cât și prin sintaxa modernă de **clase** (introdusă în ES6). Vom începe prin a explora modul în care definim **clase** și cum implementăm **moștenirea** și **compoziția**, folosind un exemplu practic. De asemenea, vom vedea cum putem **extinde prototipul** unui obiect pentru a adăuga comportamente în timpul execuției.

1.1 Clase, obiecte și moșteniri – exemplul „Robot”

Pentru a ilustra OOP în JavaScript, vom folosi un exemplu în care definim mai multe clase ce modelează roboți și arme, cu relații de moștenire și compoziție. Să examinăm codul complet mai jos, apoi îl vom explica pas cu pas:

```
class Robot {
  constructor(name) {
    this.name = name;
  }
  move() {
    console.log(`${this.name} is moving`);
  }
}

const r0 = new Robot('some robot');
r0.move();  // Output: "some robot is moving"

class Weapon {
```

```
  constructor(description) {
    this.description = description;
  }
  fire() {
    console.log(`${this.description} is firing`);
  }
}

const w0 = new Weapon('pew pew laser');
w0.fire();  // Output: "pew pew laser is firing"

class CombatRobot extends Robot {
  constructor(name) {
    super(name);
    this.weapons = [];
  }
  addWeapon(w) {
    this.weapons.push(w);
  }
  fire() {
    console.log('firing all weapons');
    for (const w of this.weapons) {
      w.fire();
    }
  }
}

const r1 = new CombatRobot('some combat robot');
r1.addWeapon(w0);
r1.fire();  // Output: "firing all weapons", then "pew pew laser is firing"

Robot.prototype.fly = function() {
  console.log(`${this.name} is flying`);
};

r1.fly();   // Output: "some combat robot is flying"
```

Explicație:

- Definim clasa **`Robot`** cu un constructor care setează un atribut `name` și cu metoda de instanță `move()`. Metoda `move` afișează un mesaj ce confirmă acțiunea robotului. După definirea clasei, instanțiem un Robot (`r0`) cu numele `"some robot"` și apelăm metoda `move()`, obținând mesajul **„some robot is moving”**. Acesta confirmă că obiectul `r0` are propria proprietate `name` și metoda moștenită din clasa sa.
- Definim clasa **`Weapon`**, care reprezintă o armă. Fiecare armă are un atribut `description` (setat în constructor) și o metodă `fire()` ce afișează **„<descriere> is firing”**. Instanțiem un obiect `w0` de tip Weapon cu descrierea `"pew pew laser"` și verificăm comportamentul `fire()`, obținând **„pew pew laser is firing”**.
- Definim clasa **`CombatRobot`**, care **extinde** (moștenește) clasa `Robot`. Sintaxa `extends Robot` indică faptul că `CombatRobot` **își va avea prototipul bazat pe prototipul** clasei `Robot`, adică un obiect de tip CombatRobot moștenește toate metodele lui Robot.
    - În constructorul lui `CombatRobot`, folosim `super(name)` pentru a apela constructorul clasei de bază (`Robot`), inițializând astfel proprietatea `name` moștenită. Apoi adăugăm proprietatea nouă `weapons` (un array) specifică roboților de luptă, inițializată ca listă goală.

- `CombatRobot` definește două metode suplimentare: `addWeapon(w)` (care adaugă o instanță de Weapon în arsenalul robotului, prin *compoziție* – robotul *conține* arme) și suprascrie metoda `fire()`. Metoda suprascrisă `fire()` afișează mai întâi un mesaj general **„firing all weapons”** apoi parcurge lista de arme și apelează metoda `fire()` a fiecăreia. Astfel, un `CombatRobot` își „cheamă” toate armele să tragă, afișând mesajele specifice armelor.
- Instanțiem un `CombatRobot` numit `r1` cu numele `"some combat robot"`. Îi adăugăm arma `w0` (creată anterior) prin `r1.addWeapon(w0)`. Când apelăm `r1.fire()`, se întâmplă următoarele:
- Metoda suprascrisă din `CombatRobot` afișează **„firing all weapons”**.
- Apoi, pentru fiecare armă din `r1.weapons` (în cazul nostru doar `w0`), se apelează `w.fire()`. Aceasta execută metoda `fire()` din clasa Weapon, afișând **„pew pew laser is firing”**. Astfel, vedem cum un obiect compus (`r1` conține `w0`) delegă comportamentul către sub-obiectul său.

- În final, observăm o caracteristică a JavaScript-ului legată de **prototipuri**: după ce am creat instanța `r1`, **adăugăm dinamic o metodă nouă** în prototipul clasei `Robot`:

```
Robot.prototype.fly = function() {
  console.log(`${this.name} is flying`);
};
```

Aceasta atașează tuturor **instanțelor** de Robot (inclusiv celor ale subclaselor, cum e `CombatRobot`) o metodă `fly()`.

Important: *în JavaScript, clasele sunt în esență funcții constructor, iar proprietățile lor prototipale (metodele) pot fi modificate runtime.*

După ce definim `Robot.prototype.fly`, putem apela `r1.fly()`. Chiar dacă `r1` a fost instanțiat **înainte** de adăugarea metodei, prin mecanismul de prototip, obiectul caută metoda în `Robot.prototype` și o găsește. Afișarea **„some combat robot is flying”** confirmă că `r1` a moștenit noua metodă.

Acest exemplu evidențiază puterea prototipurilor în JavaScript: putem extinde comportamentul claselor pe parcursul execuției (totuși, în practică, asemenea modificări ulterioare ale prototipurilor trebuie făcute cu grijă).

Sumar OOP: În acest exemplu, am evidențiat:

- **Incapsularea** datelor în clase (`Robot` știe numele său, `Weapon` știe descrierea sa, etc.).
- **Moștenirea** (`CombatRobot` extinde `Robot`, preluând atributul `name` și metoda `move()` automat, prin `extends` și `super`).
- **Polimorfismul** la nivel simplu, prin **suprascrierea** metodei `fire()` în `CombatRobot` (care înlocuiește implementarea moștenită – aici `Robot` nu avea `fire()`, dar dacă ar fi avut, `CombatRobot.fire` ar fi ascuns-o).
- **Compoziția** obiectelor (`CombatRobot` are o listă de `Weapon` – relație „are un”: un robot are arme).
- **Prototipurile și extinderea lor** – posibilitatea de a adăuga metode în `Robot.prototype` ce devin disponibile tuturor roboților existenți sau viitori.

Notă: În JavaScript modern, moștenirea se realizează astfel încât clasa derivată (`CombatRobot`) **are prototipul** setat la instanța prototip a clasei de bază (`Robot.prototype`). Astfel, toate metodele definite în `Robot.prototype` (cum este `move()`, și ulterior `fly()`) sunt accesibile prin lanțul de prototipuri oricărui `CombatRobot`. Fiecare instanță (precum `r1`) are un prototip intern care referă `CombatRobot.prototype`, care la rândul său referă `Robot.prototype`. Dacă se caută o metodă neexistentă pe `r1` (de exemplu `fly` înainte să fie definit), JavaScript urcă pe lanț: nu găsește în `CombatRobot.prototype` (căci nu era definită acolo inițial), apoi caută în `Robot.prototype`. După ce adăugăm `fly` în `Robot.prototype`, acea metodă devine vizibilă și pentru `r1`.

Exercițiu de reflecție:

*Ce se întâmplă dacă apelăm `r0.fly()` (unde `r0` este un simplu Robot) după adăugarea metodei în prototip? Încearcă să anticipezi răspunsul și apoi testează mental: `r0.fly()` va funcționa, deoarece și `r0` (fiind instanță a lui Robot) va găsi metoda `fly` în `Robot.prototype`, afișând **„some robot is flying”**.*

1.2 Prototipuri vs. clase – detalii importante

Exemplul precedent folosește sintaxa de **clasă** introdusă în ES6, care este în esență o „îmbrăcăminte” sintactică peste sistemul de prototipuri al JavaScript. Intern, o clasă definește automat proprietățile prototipale (metodele) și pe cele de instanță (proprietățile setate în constructor). Câteva aspecte de reținut:

- **Atributele de instanță** (precum `this.name`, `this.description`, `this.weapons`) sunt stocate direct în obiectul instanțiat (`r0`, `w0`, `r1`). Fiecare obiect are propriile valori pentru acestea.
- **Metodele** definite în interiorul clasei (precum `move`, `fire`) sunt stocate o singură dată, în prototipul clasei (`Robot.prototype`, `Weapon.prototype`, etc.), și **nu sunt duplicate** pentru fiecare instanță. Obiectele instanță **deleagă** apelurile acestor metode către prototip.
- **Moștenirea prototipală**: `CombatRobot.prototype` este legat de `Robot.prototype`. De aceea, chiar dacă `Robot.prototype` capătă ulterior o metodă nouă (`fly`), automat `CombatRobot.prototype` o „vede” deoarece lanțul de prototipuri duce acolo.
- Putem adăuga metode nu doar în prototipul claselor noastre, dar și în prototipul obiectelor nativ existente. De exemplu, am fi putut face `String.prototype.firstChar = function() { return this.charAt(0); }` pentru a extinde obiectele de tip șir de caractere cu o nouă funcționalitate (nu recomandăm modificarea prototipurilor built-in în producție, dar ca experiment academic e util).

1.3 Moștenirea și agregarea – continuare

Moștenirea permite unui tip de obiect să preia automat proprietăți și comportamente ale altuia. În exemplul nostru, `CombatRobot` moștenește de la `Robot` astfel încât nu a trebuit să redefinim `move()` sau `name`; le-am preluat. Moștenirea se folosește pentru a modela o relație de tip **„este un”** (CombatRobot *este un* Robot, deci are toate trăsăturile de bază ale robotului, plus altele specifice).

Compoziția (sau ***agregarea***) presupune că un obiect conține alt obiect ca parte componentă. Relația este de tip **„are un”**: CombatRobot *are* Weapon (are arme). Folosim compoziția pentru a reutiliza funcționalitate sau a modela relații parte-întreg fără a folosi moștenirea. În exemplul dat, nu ar fi avut sens ca `CombatRobot` să *moștenească* `Weapon` – un robot nu *este* o armă, ci are una. Astfel, relația corectă e compoziție: robotul conține o listă de arme.

Extinderea prototipurilor: După cum am arătat cu `Robot.prototype.fly`, JavaScript ne permite să schimbăm dinamica de **disponibilitate a metodelor** la runtime. În alte limbaje OOP, adăugarea de metode noi la o clasă *după* ce au fost create obiecte nu este posibilă sau trivială. În JS, prototipurile oferă flexibilitate mare. Acest lucru poate fi util pentru polifill-uri (adăugarea de funcționalități lipsă pe obiectele built-in în browsere mai vechi), însă trebuie folosit cu precauție ca să nu introducă conflicte (de exemplu, suprascrierea accidentală a unei metode existente în prototip).

Exercițiu propus:

*Pornind de la exemplul de mai sus, **implementați următoarea ierarhie de clase**: o clasă `Software` cu metoda `run()`. Să se definească o clasă `Browser` care **moștenește** `Software` și permite instalarea unor `Plugin`-uri. Fiecare `Browser` poate avea mai multe instanțe de `Plugin` (clasă separată). Gândiți-vă la relațiile potrivite: un Browser este un Software (moștenire), un Browser are Plugin-uri (compoziție). Implementați `Browser.install(plugin)` pentru a adăuga un plugin, și eventual o metodă `Browser.listPlugins()` pentru afișarea plugin-urilor instalate. Astfel de exerciții vă ajută să aplicați conceptele de moștenire și compoziție.*

2. Funcții recursive și memoizare – optimizare prin tehnici OOP/FP

În această secțiune ne vom abate puțin de la OOP clasic pentru a discuta un concept legat atât de **eficiența algoritmilor**, cât și de organizarea inteligentă a codului: **memoizarea**. Vom folosi ca exemplu calculul numerelor Fibonacci. Acesta este un exemplu clasic unde o abordare recursivă simplă este ineficientă, dar se poate optimiza memorând rezultatele deja calculate. Deși nu implică direct clase și moștenire, această tehnică implică noțiuni de **încapsulare** a datelor și de **utilizare a unor closure** (închideri) – concepte ce țin de modul în care funcțiile pot proteja date (similar într-un fel cu obiectele care își protejează atributele).

2.1 Recursivitate naivă vs. memoizare (exemplul Fibonacci)

Șirul Fibonacci este definit astfel: *F(0)=0, F(1)=1, iar pentru n ≥ 2, F(n) = F(n-1) + F(n-2)*. O implementare recursivă directă a funcției Fibonacci în JavaScript arată astfel:

```
function fib(n) {
  if (n < 2) return n;
  return fib(n - 1) + fib(n - 2);
}
```

Această funcție are un mare dezavantaj: recalculează de multe ori aceleași valori intermediare. Complexitatea sa temporală crește exponențial cu n (mai precis, ~ϕ^n unde $\phi \approx 1.618$). De exemplu, `fib(40)` face sute de milioane de apeluri recursive, ceea ce este ineficient.

***Memoizarea** (memorization) este o tehnică prin care **memorăm rezultatele** funcției pentru anumite argumente, astfel încât dacă funcția e chemată din nou cu același argument, să returneze instant rezultatul din memorie, în loc să recalculeze. Acest procedeu folosește o structură (de obicei **un obiect** sau **dicționar**) pentru a păstra perechi argument->rezultat deja compute.*

Vom crea o versiune optimizată `fibMemo(n)` folosind memoizare. O metodă elegantă în JavaScript este să folosim o **funcție imediat invocată** (IIFE) care să creeze un mediu închis (closure) ce conține un obiect de memoizare:

```
const fibMemo = (() => {
  const memo = { 0: 0, 1: 1 };      // cazurile de bază memorate
  return function f(n) {
    if (n in memo) {
```

```
      return memo[n];              // întoarce direct dacă am calculat înainte
    }
    // altfel, calculează și memorează rezultatul
    memo[n] = f(n - 1) + f(n - 2);
    return memo[n];
  };
})();
```

Să explicăm această construcție pas cu pas:

- Definim `fibMemo` ca fiind rezultatul unei funcții imediat apelate (`(() => { ... })()`). În interiorul acesteia, creăm un obiect `memo` care joacă rol de cache (inițializăm cu valorile cunoscute pentru 0 și 1). Returnăm **funcția propriu-zisă** `f(n)` care realizează calculul cu verificarea cache-ului.
- Când `fibMemo` este creat, codul din IIFE se execută o singură dată, definind `memo` și returnând funcția `f`. De acum înainte, `fibMemo` este chiar referința către funcția internă `f(n)` *care reține referința la obiectul `memo` închis în contextul ei*.
- La fiecare apel `fibMemo(x)`, funcția verifică dacă `memo` conține deja rezultatul pentru `x`. Dacă da, nu recalculează, ci returnează direct. Dacă nu, calculează recursiv (apelându-se pe sine pentru subprobleme mai mici), **stochează** rezultatul în `memo[x]`, apoi îl returnează.
- Avantajul major: fiecare valoare F(n) va fi calculată **o singură dată**. Complexitatea devine liniară O(n) (practic 2n operații: calculează de la 2 până la n, memorând pe parcurs). `memo` acționează ca o *memorie persistentă* între apelurile recursiv interne, datorită closure-ului.

Iată un scurt exemplu de utilizare pentru a vedea diferența de performanță:

```
console.time('fib40');
console.log( fib(40) );      // calculează recursiv naiv
console.timeEnd('fib40');
================================================================================
console.time('fibMemo40');
console.log( fibMemo(40) );  // calculează cu memoizare
console.timeEnd('fibMemo40');
```

Dacă ați rula acest cod, veți observa că `fib(40)` durează considerabil mai mult timp decât `fibMemo(40)`, de ordinul secundelor, pe când cel memoizat va răspunde aproape instantaneu. Acest test mental evidențiază eficiența memoizării.

Observație OOP/organizare: În exemplul de mai sus, am folosit un obiect simplu `memo` pentru a stoca valori. O abordare alternativă ar fi fost să definim `fibMemo` ca metodă statică a unei clase sau să folosim o clasă cu o instanță care conține un astfel de cache. De exemplu, se putea crea o clasă `FibonacciCalculator` care să aibă un membru `memo` și o metodă `fib(n)` ce folosește `this.memo`. În JavaScript însă, adesea se preferă soluții mai simple funcționale, cum este closure-ul de mai sus, care **încapsulează** complet logica și datele (similar cu un obiect, dar fără sintaxa unei clase).

2.2 Validarea intrării și aruncarea de excepții

Pentru a completa scriptul nostru Fibonacci, să ne imaginăm că vrem să folosim această logică într-un program Node.js care primește `n` ca parametru din linia de comandă și afișează F(n). Trebuie să ne asigurăm că intrarea este validă (un număr întreg ne-negativ) și, dacă nu, să **semnalăm o eroare**.

Putem adăuga o funcție utilitară:

```
function parseNonNegInt(x) {
  const n = Number(x);
  if (!Number.isInteger(n) || n < 0) {
    throw new TypeError('Trebuie un număr întreg nenegativ');  // aruncăm excepție
dacă invalid
  }
  return n;
}
```

Această funcție încearcă să convertească valoarea x la număr și verifică dacă e întreg și nenegativ. Dacă nu, aruncă o eroare de tip `TypeError` cu un mesaj descriptiv.

Unind toate piesele, scriptul nostru complet (să-l numim `fibo_mem.js`) ar putea arăta astfel:

```
const [,, arg] = process.argv;
if (arg === undefined) {
  console.error("Utilizare: node fibo_mem.js <n>");
  process.exit(1);
}

function parseNonNegInt(x) {
  const n = Number(x);
  if (!Number.isInteger(n) || n < 0) {
    throw new TypeError("Trebuie un număr întreg nenegativ.");
  }
  return n;
}

const fibMemo = (() => {
  const memo = {0: 0, 1: 1};
  return function f(n) {
    if (n in memo) {
      return memo[n];
    }
    memo[n] = f(n-1) + f(n-2);
    return memo[n];
  };
})();

try {
  const n = parseNonNegInt(arg);
  console.log(`Fib(${n}) = ${fibMemo(n)}`);
} catch (e) {
  console.error("Eroare:", e.message);
  process.exit(1);
}
```

Ce face acest script:

- Folosește `process.argv` pentru a prelua argumentul trimis scriptului (ignorând primele două elemente care sunt calea către Node și calea către script).
- Dacă nu s-a furnizat niciun argument, afișează un mesaj de utilizare și iese cu cod de eroare. - Definește funcția `parseNonNegInt` pentru validarea intrării.
- Definește `fibMemo` (ca mai sus) pentru calcul optimizat.
- În blocul `try...catch`, încearcă să parseze argumentul și să calculeze Fibonacci. Dacă `parseNonNegInt` aruncă o eroare (de exemplu s-a dat "abc" sau "-5"), aceasta este prinsă în `catch`, se afișează mesajul de eroare și se iese cu cod de eroare.
- Dacă totul e OK, se afișează rezultatul, de forma **Fib(n) = X**.

De reținut:

- Am folosit intenționat **aruncarea de excepții** (`throw new TypeError`) pentru a ilustra cum putem gestiona erorile în stil OOP/structurat. În Node.js, un alt mod comun este să transmiți erori prin callback-uri sau să returnezi un `Promise` respins. Dar aici, fiind un script procedural simplu, `throw` este adecvat, iar `try/catch` capturează elegant problema.
- `TypeError` este un tip de eroare nativ JavaScript care semnalează că a fost întâlnit un tip nepotrivit (în cazul nostru, se aștepta un număr întreg, dar am primit altceva).
- Utilizarea lui `process.exit(1)` indică sistemului de operare că programul s-a încheiat cu eroare (codul de ieșire 0 indică succes, valori nenule indică eroare). Acest lucru este util dacă folosim scriptul în contexte automatizate – de exemplu, un alt proces poate verifica codul de ieșire.

Legătura cu OOP: Deși în acest script nu am definit clase, am folosit:

- **Încapsularea prin closure** (similar cu încapsularea într-un obiect) pentru a menține `memo` ascuns funcției `fibMemo`.
- Conceptul de **stare internă** – `memo` joacă rol de stare internă care persistă de la un apel al funcției la altul, la fel cum un obiect își păstrează atributele între apeluri ale metodelor.
- **Excepțiile** – care sunt o mecanică de tratament al erorilor folosită intens în OOP. Aruncând și prindând excepții, ne asigurăm că erorile sunt semnalate clar și pot fi tratate la un nivel superior al codului, separat de logica de calcul (principiu de separare a preocupărilor).

Exercițiu:

Modificați funcția `fibMemo` de mai sus pentru a număra de câte ori este apelată recursiv (de fapt, de câte ori calculează efectiv o valoare nouă). ***Hints:*** *puteți adăuga o variabilă globală sau externă în closure care să incrementeze ori de câte ori se intră în ramura ne-memorată. Comparați apoi numărul de apeluri pentru `fib(10)` (fără memoizare) cu `fibMemo(10)`. Veți observa diferența dramatică în numărul de operații.*

3. Programare asincronă în Node.js

Node.js este construit în jurul unui ***model asincron, bazat pe evenimente***, ceea ce înseamnă că multe operații (în special cele de I/O – citire/scriere fișiere, rețea, etc.) nu blochează execuția programului în timp ce așteaptă finalizarea. În schimb, acestea pornesc operația și **programează** o funcție callback sau returnează un ***promise*** ce va fi rezolvat când operația se termină. Între timp, scriptul poate continua cu alte operații. Acest mecanism este crucial pentru eficiența Node.js, permițând unui singur fir de execuție să gestioneze mii de conexiuni fără a aștepta inactiv.

Vom explora conceptul de asincronism prin **trei exemple**:

1. **Stream-uri** (fluxuri) Node.js – modul în care putem citi și scrie date în mod asincron, pe bucăți.

2. Un scenariu pas cu pas „**orderCoffee**” – care simulează pașii în pregătirea unei cafele folosind temporizări pentru a evidenția ordinea execuției în cod asincron (vom folosi **callback-uri** sau promisiuni).

3. **Citirea de fișiere** – comparând varianta **sincronă** cu cea **asincronă** (prin callback), pentru a înțelege diferența de comportament.

3.1 **Stream-uri** în Node.js – exemplul `S4v01stream`

Un **stream** (flux) în Node.js este o abstracție pentru citirea sau scrierea continuă a datelor, bucățică cu bucățică, în loc de a procesa totul odată. Fluxurile pot fi de mai multe tipuri: Readable (doar de citire), Writable (scriere), sau Duplex/Transform (ambele). Un exemplu comun este citirea unui fișier text foarte mare: în loc să încărcăm întreg fișierul în memorie (ceea ce ar consuma mult RAM și ar bloca Node până s-a terminat citirea), putem citi treptat datele pe măsură ce devin disponibile și procesa progresiv.

SCENARIU: Să spunem că avem un fișier text și dorim să îl afișăm la consolă sau să-l scriem într-un alt fișier, cu eventuale transformări (de exemplu, să numerotăm liniile sau să schimbăm literele mici în mari). Vom folosi un stream pentru citire și, opțional, unul de scriere, demonstrând programarea asincronă: Node va citi datele **în fundal** și ne va anunța prin evenimente pe măsură ce segmente din fișier au fost citite.

Iată un exemplu simplu de script Node (să-l numim `stream_read.js`) care citește dintr-un fișier și scrie conținutul în consolă, folosind fluxuri:

```
const fs = require('fs');
const filePath = 'input.txt';  // un fișier text existent

const readStream = fs.createReadStream(filePath, { encoding: 'utf8' });

readStream.on('data', chunk => {
  console.log('<< Chunk primit:', chunk.length, 'caractere >>');
  process.stdout.write(chunk);  // scriem direct bucata în consola
});

readStream.on('end', () => {
  console.log('\n*** Sfârșit de fișier ***');
});

readStream.on('error', err => {
  console.error('Eroare la citire:', err.message);
});
```

Cum funcționează acest cod:

Folosim `fs.createReadStream` din modulul `fs` (File System) pentru a obține un stream de citire din fișierul specificat. Îi furnizăm opțiunea `{ encoding: 'utf8' }` astfel încât datele citite să fie furnizate direct ca șiruri de caractere (și nu ca buffer binar). - `readStream` emite evenimentul `'data'` de fiecare dată când o bucată de date este disponibilă. Ne abonăm la acest eveniment folosind `readStream.on('data', chunk => { ... })`. Callback-ul va primi ca argument `chunk` – un șir de caractere (partea din fișier citită).

În interior:

- Afișăm un mesaj debug (nu obligatoriu, dar util de observat) care indică mărimea bucății primite.
- Scriem efectiv conținutul bucății în `stdout` (consolă) folosind `process.stdout.write`. (Folosim `write` în loc de `console.log` pentru a evita adăugarea unui newline implicit, astfel conținutul fișierului se va afișa exact cum este.)

- Ne abonăm la evenimentul `'end'`, care este emis o singură dată, când stream-ul ajunge la sfârșitul fișierului. Acolo putem afișa un mesaj final (de exemplu, " ***Sfârșit de fișier*** ").
- De asemenea, ascultăm la evenimentul `'error'` în caz că apar probleme (fișierul nu există, permisiuni insuficiente etc.), și raportăm eroarea.

Comportament ASINCRON: Observați că programul *nu așteaptă explicit* citirea fișierului. Imediat după ce apelează `fs.createReadStream`, Node începe să citească date de pe disc în fundal. Între timp, scriptul principal poate continua. În cazul nostru, scriptul se pune practic în așteptare, menținând procesul viu doar prin faptul că stream-ul deschis are evenimente nerezolvate. Pe măsură ce kernelul furnizează date, Node declanșează callback-urile `'data'` de mai multe ori. Astfel, textul se afișează ***pe măsură ce este citit***, fără să încărcăm totul în memorie. Dacă fișierul e foarte mare, vom vedea în consolă cum "curge" treptat tot conținutul.

Putem extinde exemplul să demonstreze și un stream de scriere. De exemplu, să citim din `input.txt` și să scriem totul cu litere mari într-un `output.txt`. Pentru aceasta, vom folosi un stream de scriere și un transform stream intermediar (o componentă care transformă datele trecându-le mai departe).

Un **transform-stream** se poate crea fie manual (extinzând `stream.Transform`), fie putem folosi metoda simplă `pipeline` din modulul `'stream'` pentru a lega mai multe stream-uri. Un exemplu minimal folosind `pipeline` și fluxuri existente:

```
const { createReadStream, createWriteStream } = require('fs');
const { Transform, pipeline } = require('stream');

const upperCaseTr = new Transform({
  transform(chunk, encoding, callback) {
    // transformă chunk-ul în string și apoi în majuscule
    const output = chunk.toString().toUpperCase();
    callback(null, output);
  }
});

pipeline(
  createReadStream('input.txt'),
  upperCaseTr,
  createWriteStream('output.txt'),
  err => {
    if (err) {
      console.error('Pipeline failed:', err);
    } else {
      console.log('Pipeline reușit. Datele transformate au fost salvate în
output.txt');
    }
  }
);
```

În acest exemplu (ceva mai complex):

- Creăm un stream de transformare `upperCaseTr` care implementează funcția `transform` – ce primește bucăți de date, le transformă (în majuscule) și apelează `callback` cu rezultatul.
- Folosim `pipeline` pentru a lega: stream-ul de citire -> transform -> stream-ul de scriere. `pipeline` gestionează automat înscrierea fiecărui chunk transformat în stream-ul de scriere și propagarea erorilor.
- La final, în callback-ul lui `pipeline`, aflăm dacă totul a fost ok sau s-a produs o eroare.

DE CE SUNT STREAM-URILE ASINCRONE? *Operațiile de I/O (citire de pe disc, rețea etc.) sunt mult mai lente decât operațiile CPU. Node.js folosește mecanisme asincrone (non-blocante) pentru I/O: pornește operația și continuă execuția altor lucruri; când I/O e gata (citit un chunk, de exemplu), primește o notificare și rulează callback-ul asociat. Astfel, un singur thread poate procesa simultan multe lucrări de I/O fără să stea degeaba.*

3.2 Ordonarea execuției în cod asincron – exemplul `orderCoffee`

Pentru a înțelege **ordonarea instrucțiunilor** în prezența asincroniei, vom crea un scenariu fictiv: prepararea unei cafele la un automat, cu mai mulți pași, unii care durează un timp (simulați cu `setTimeout`). Vom vedea cum codul scris secvențial *nu* execută neapărat acțiunile în ordinea scrierii lor, dacă unele sunt asincrone.

Scenariul "cafeaua":

1. Plasarea comenzii de cafea (`placeOrder`).
2. Fierberea apei (`boilWater`) – durează, să zicem, 2 secunde.
3. Prepararea cafelei (`brewCoffee`) – durează 1 secundă.
4. Servirea cafelei (`serveCoffee`).

Dacă am scrie codul în mod **sincron**, ar fi simplu în ordinea de mai sus. În mod asincron, vom simula timpii de așteptare cu `setTimeout` și vom folosi fie **callback-uri înlănțuite**, fie **Promise-uri/async-await** pentru a ne asigura că ordinea logică este respectată.

3.2.1 Varianta cu callback-uri (piramida apelurilor)

```
function placeOrder(customer, callback) {
  console.log(`Preiau comanda de cafea de la ${customer}...`);
  setTimeout(() => {
    console.log(`Comanda pentru ${customer} a fost înregistrată.`);
    callback();  // trecem la pasul următor după 1s
  }, 1000);
}

function boilWater(callback) {
  console.log("Încep să fierb apa...");
  setTimeout(() => {
    console.log("Apa a fiert.");
    callback();
  }, 2000);
}

function brewCoffee(callback) {
  console.log("Prepar cafeaua...");
  setTimeout(() => {
    console.log("Cafeaua e preparată.");
    callback();
  }, 1000);
}

function serveCoffee(customer) {
  console.log(`Servesc cafeaua pentru ${customer}. Poftă bună?`);
}

// Secvența de preparare folosind callback-uri
placeOrder("Alice", () => {
```

```
  boilWater(() => {
    brewCoffee(() => {
      serveCoffee("Alice");
    });
  });
});
```

Ce face acest cod:

- `placeOrder(customer, callback)`: simulează înregistrarea comenzii; după 1 secundă (folosind `setTimeout`), apelează `callback()` indicând că poate continua fluxul.
- `boilWater(callback)`: fierbe apa (mesaj inițial, apoi după 2 secunde mesaj de final și apel al callback-ului).
- `brewCoffee(callback)`: similar, prepară cafeaua în 1 secundă, apoi callback.
- `serveCoffee(customer)`: acțiune finală sincronă (doar afișează servirea).

La rulare, ordinea afișărilor ar fi:

```
Preiau comanda de cafea de la Alice...
Comanda pentru Alice a fost înregistrată.
Încep să fierb apa...
Apa a fiert.
Prepar cafeaua...
Cafeaua e preparată.
Servesc cafeaua pentru Alice. Poftă bună?
```

Observăm că am reușit să păstrăm ordinea logică dorită, **dar** structura codului este una de „piramidă” de callback-uri, adesea numită ***callback hell*** (iadul callback-urilor). Fiecare nivel interior este indentat, ceea ce devine greu de citit și de gestionat dacă avem mulți pași sau dacă trebuie să tratăm erori la fiecare nivel. De exemplu, dacă `boilWater` ar avea un posibil eșec, ar trebui să transmitem un `err` parametru în callback și să adăugăm ramuri de if pentru erori la fiecare nivel, complicând codul.

3.2.2 Varianta cu Promises și `async/await` (stil modern și mai lizibil)

O alternativă modernă este să folosim **Promise**-uri. `setTimeout` nu are nativ suport pentru promisiuni, dar îl putem transforma manual într-un promise pentru a-l `await`-a. Mai simplu, vom modifica funcțiile noastre să returneze promisiuni în loc să folosească callback-uri:

```
function placeOrder(customer) {
  console.log(`Preiau comanda de cafea de la ${customer}...`);
  return new Promise(resolve => {
    setTimeout(() => {
      console.log(`Comanda pentru ${customer} a fost înregistrată.`);
      resolve();  // continuă după 1s
    }, 1000);
  });
}

function boilWater() {
  console.log("Încep să fierb apa...");
  return new Promise(resolve => {
    setTimeout(() => {
      console.log("Apa a fiert.");
      resolve();
    }, 2000);
  });
```

```
}

function brewCoffee() {
  console.log("Prepar cafeaua...");
  return new Promise(resolve => {
    setTimeout(() => {
      console.log("Cafeaua e preparată.");
      resolve();
    }, 1000);
  });
}

function serveCoffee(customer) {
  console.log(`Servesc cafeaua pentru ${customer}. Poftă bună!`);
}

// Secvența de preparare folosind async/await
async function prepareCoffeeFor(customer) {
  await placeOrder(customer);
  await boilWater();
  await brewCoffee();
  serveCoffee(customer);
}

prepareCoffeeFor("Alice");
```

În această variantă:

- Fiecare funcție (mai puțin `serveCoffee` care nu are așteptare) **returnează un `Promise`** care va fi rezolvat după încheierea acțiunii simulate. Folosim `new Promise(resolve => { ... resolve() ... })` pentru a construi un promise manual care rezolvă când e gata. (În cod real, multe funcții de I/O returnează direct promise-uri – ex. dacă am folosi `fs.promises.readFile`, am primi un promise și am putea `await` direct.)
- Funcția `prepareCoffeeFor` este marcată `async`, permițând folosirea cuvântului cheie `await` în interior. Aceasta face codul să arate aproape sincronic, deși execuțiile sunt asincrone:
    - `await placeOrder(customer)` va apela `placeOrder` (care pornește timerul de 1s și returnează imediat un promise neîndeplinit încă). `await` va **suspenda** execuția lui `prepareCoffeeFor` **fără** a bloca firul de execuție, până când promise-ul este rezolvat (după 1s). Între timp, alte operații pot avea loc (în exemplul simplu nu sunt alte sarcini, dar în servere Node ar putea deservi alt client).
    - După 1s, când `resolve()` este apelat în `placeOrder`, execuția în `prepareCoffeeFor` reia de unde a rămas, trece la următoarea linie.
    - Similar `await boilWater()` (așteaptă 2s), apoi `await brewCoffee()` (1s).
- Când toate astea s-au terminat, se execută `serveCoffee(customer)`.
- Rezultatul în consolă este același ca înainte, dar codul este mult mai **linear și lizibil**. Practic am eliminat indentările excesive și am scos în evidență succesiunea logică.

Notă: *În spatele sintaxei, Node continuă să ruleze programul asincron. `await` nu blochează firul – el doar face ca secvența din `prepareCoffeeFor` să aștepte promise-ul, dar motorul de runtime poate executa alt cod între timp.*

CONCLUZIE (*la exemplul de asincronism*): Atât varianta cu callback, cât și cea cu `async/await` realizează același lucru: se asigură că pașii se execută în ordinea dorită, chiar dacă unii sunt amânați. Diferența constă în modul de organizare a codului și tratarea erorilor:

- Cu callback-uri, erorile se tratează de obicei prin convenție (ex. primul parametru al callback-ului e `err` dacă există). Cu promise-uri/async, putem folosi `try...catch` în interiorul unei funcții async pentru a prinde eventuale erori aruncate sau promise-uri respinse.
- `async/await` și promise-urile fac codul asincron să semene cu cod sincron, îmbunătățind claritatea și fiind mai ușor de întreținut.

Exercițiu:

Rescrieți secvența de preparare a cafelei fără a folosi `async/await`, ci lucrând direct cu promisiuni:

```
placeOrder("Alice")
  .then(() => boilWater())
  .then(() => brewCoffee())
  .then(() => serveCoffee("Alice"))
  .catch(err => console.error("Eroare in fluxul cafelei:", err));
```

Aceasta este abordarea *then/catch* echivalentă. Observați cum și aceasta evită indentarea în lanț, însă folosește lanțul de `.then()`. Comparați cu varianta `async/await` și decideți care e mai ușor de urmărit pentru voi.

3.3 Operații sincrone vs. asincrone – exemplul citirii unui fișier (`country_bounds`)

Ultimul exemplu compară două abordări pentru aceeași sarcină: să citim conținutul unui fișier (de exemplu, un fișier JSON cu coordonatele frontierelor unei țări – de aici numele `country_bounds`). Vom vedea versiunea **sincronă** (blocantă) și versiunea **asincronă** (neblocantă) a codului, pentru a evidenția diferențele.

Să presupunem că avem un fișier `country_bounds.json` de câteva megabytes cu date geografice. Dorim să citim acest fișier și apoi să calculăm, de exemplu, câte caractere are (sau să extragem o informație). Vom insera și niște `console.log`-uri pentru a vedea ordinea execuției.

VARIANTA A (*sincronă*):

```
const fs = require('fs');

console.log("Citire sincronă început...");
try {
  const data = fs.readFileSync('country_bounds.json', 'utf8');
  console.log("Fișier citit (sincron), lungime:", data.length);
} catch (err) {
  console.error("Eroare la citirea fișierului (sync):", err);
}
console.log("După citire sincronă.");
```

Ce face codul sincron de mai sus:

- `fs.readFileSync` citește întreg fișierul **în mod blocant**. Asta înseamnă că **niciun alt cod nu se execută în acest timp**; Node nu va putea procesa alte evenimente (dacă ar fi), deoarece firul de execuție este ocupat așteptând citirea de pe disc.
- Dacă fișierul este mare, această operație poate dura zeci de milisecunde (sau mai mult, depinde de disc). În acel interval, programul stă blocat.

- Odată terminată citirea, funcția returnează conținutul (`data`) ca șir de caractere (deoarece am specificat `utf8`).
- Afișăm lungimea datelor citite și eventual un mesaj final. - Notăm că am folosit un bloc `try...catch` pentru a prinde posibila excepție aruncată de `readFileSync` în caz de eroare (când se întâmplă o eroare, varianta sincronă *aruncă* excepție).
- Ordinea mesajelor în consolă ar fi: 1. "Citire sincronă început..." 2. (pauză până se termină citirea fișierului de pe disc) 3. "Fișier citit (sincron), lungime: X" 4. "După citire sincronă."

VARIANTA B (*asincronă, cu callback*):

```
console.log("Citire asincronă început...");

fs.readFile('country_bounds.json', 'utf8', (err, data) => {
  if (err) {
    console.error("Eroare la citirea fișierului (async):", err);
    return;
  }
  console.log("Fișier citit (asincron), lungime:", data.length);
});

console.log("După apelul asincron fs.readFile.");
```

Ce face codul asincron:

- Apelează `fs.readFile` (variantă asincronă) cu un callback. Imediat după apel, **controlul revine** programului
- Node începe operația de citire în fundal, dar *nu așteaptă*.
- Mesajul "După apelul asincron..." va fi afișat **înainte** ca fișierul să fie citit complet, practic imediat după pornirea citirii. Astfel, vom vedea în consolă: 1. "Citire asincronă început..." 2. "După apelul asincron fs.readFile." 3. (după un timp, când citirea se finalizează) "Fișier citit (asincron), lungime: X"
- Observăm deci că **execuția principală nu așteaptă** rezultatul citirii; ultimul `console.log` din cod este executat imediat. Când datele devin disponibile, Node apelează funcția noastră callback, care afișează lungimea.
- În callback, verificăm `err` – la apelurile asincrone în stil callback, eroarea este indicată ca prim argument. Dacă e ne-null, logăm eroarea și ieșim (`return` din callback). Dacă nu e eroare, `data` conține conținutul fișierului și putem lucra cu el (în exemplu, doar afișăm lungimea).

- **AVANTAJ:** În perioada de așteptare a citirii, thread-ul poate face altceva (dacă ar exista alt cod de executat sau alte evenimente de gestionat). În exemplul nostru punem doar un `console.log`, dar imaginați-vă un server care poate răspunde altor cereri între timp.
- **CAPCANA:** Orice cod care depinde de `data` trebuie să fie în interiorul callback-ului (sau să fie apelat din el). Dacă am încerca să folosim `data` imediat după apelul `fs.readFile`, nu am avea încă rezultatul (*de aceea am și mutat logica de utilizare în interiorul callback-ului*).

COMPARAREA CELOR DOUĂ VARIANTE (A vs. B):

- În varianta **sincronă**, codul este liniar, ușor de citit, dar **blochează complet** aplicația pe durata I/O. Dacă am rula acest cod pe un server web Node cu multiple cereri, nicio

altă cerere nu ar fi procesată în timpul citirii fișierului – ceea ce degradează mult performanța și experiența utilizatorilor.

- În varianta **asincronă**, codul e puțin mai complicat (trebuie să punem logica dependentă de date în callback, sau să folosim promise-uri/async-await pentru a-l face mai liniar), însă aplicația **rămâne receptivă**. Node poate servi alte evenimente sau cereri în paralel cu citirea fișierului. Aceasta este metoda preferată pe server – nu vrem ca un singur I/O mai lent să blocheze restul.

ALTERNATIVA cu `fs.promises` și async/await: Node oferă și un modul `fs/promises` care expune metode ce returnează promise-uri. Astfel, putem scrie ***echivalent asincron*** astfel:

```
const fsp = require('fs/promises');
async function readFileAsync() {
  try {
    console.log("Citire promisă început...");
    const data = await fsp.readFile('country_bounds.json', 'utf8');
    console.log("Fișier citit (await), lungime:", data.length);
  } catch (err) {
    console.error("Eroare la citire (await):", err);
  }
  console.log("După await la citire.");
}
readFileAsync();
console.log("Aceasta linie poate aparea inaintea citirii complete.");
```

Dacă executați mental, veți vedea:

- "Citire promisă început..."
- Imediat: "Aceasta linie poate aparea inaintea citirii complete." (deoarece readFileAsync este lansată dar așteaptă I/O)
- Apoi când fișierul e citit: "Fișier citit (await), lungime: X" - "După await la citire."

Aceasta combină lizibilitatea codului sincron cu avantajele asincroniei. Într-o funcție `async`, putem citi fișierul cu `await` fără a bloca thread-ul principal.

DE CE SĂ PREFERĂM CODUL NEBLOCANT?

În Node.js, o buclă de evenimente unică gestionează tot. Dacă blocăm această buclă (de exemplu cu `readFileSync` sau cu un loop imens calculând ceva), nicio altă parte a aplicației nu mai poate rula între timp. Pentru aplicații server sau aplicații care trebuie să rămână interactive, acest lucru e problematic. De aceea, Node încurajează puternic folosirea metodelor asincrone. Există cazuri unde codul sincron e acceptabil (scripturi mici de administrație, scripturi CLI scurte unde simplitatea contează mai mult decât performanța), dar ***e important să cunoaștem ambele abordări și implicațiile lor***.

3.4 Evenimentul `loop` și executarea concurentă

(**FACULTATIV PENTRU APROFUNDARE**) Merită menționat cum funcționează Node "sub capotă": există un **Event Loop** (bucla de evenimente) care are mai multe faze. Operațiile asincrone (I/O, timer-e, etc.) înregistrează callback-urile în bucla de evenimente. Bucla rulează continuu, verificând: *sunt callback-uri gata de executat?* Dacă da, le execută unul câte unul. Dacă nu, stă în așteptare. Fiecare iterație a buclei ia un callback pregătit, îl execută complet (până la capăt sau până când acesta programează altele) apoi revine să verifice următorul eveniment. Acest mecanism face ca, deși Node

e single-thread, să pară că face multitasking, deoarece intercalează execuțiile diferitelor operații **fără blocaje**.

Un reper important: **orice cod sincron CPU-intensiv** (de exemplu, un calcul al numerelor prime foarte mari, un loop imens) va bloca totul până termină. Asemenea sarcini ar trebui fie optimizate, fie ***mutate în thread-uri de lucru separate*** (Node permite folosirea de ***Worker Threads*** pentru calcule paralele, dacă e necesar, ***sau delegarea către procesoare C++ native***).

Exercițiu:

Pentru a vizualiza blocarea buclei de evenimente, puteți încerca un experiment: scrieți un cod care pornește un `setTimeout` de 1 sec. apoi execută un loop intens de ~2 secunde:

```
setTimeout(() => console.log("Timeout 1s ajuns!"), 1000);
let sum = 0;
for(let i=0; i<1e9; i++){ sum += i; }  // loop mare ce durează >1s
console.log("Loop gata, suma este", sum);
```

Rulați-l mental: deși timer-ul era setat la 1s, mesajul "Timeout 1s ajuns!" va apărea **după** ce loop-ul își termină execuția (~2s). Aceasta pentru că bucla de evenimente nu poate executa callback-ul timerului până nu termină codul sincron (loop-ul) aflat deja în curs.

CE AM ÎNVĂȚAT – RECAPITULARE

În această lecție am parcurs concepte cheie despre **OOP în JavaScript** și **programare asincronă în Node.js**:

- **Clase și Prototipuri:** Am văzut cum definim clase în JS, cum folosim `extends` și `super` pentru moștenire, precum și modul în care metodele sunt distribuite prin prototipuri. Exemplul cu roboți a evidențiat moștenirea (Robot -> CombatRobot), compoziția (CombatRobot conține Weapons) și chiar extinderea prototipurilor în timpul execuției (adăugarea metodei `fly`).
- **Încapsulare și reutilizare:** Atât prin clase cât și prin funcții cu closure (ex: `fibMemo`), JavaScript ne permite să încapsulăm date (proprietăți de obiect sau variabile închise într-o funcție) și să le protejăm de spațiul global, oferind în același timp metode de acces controlat.
- **Memoizare:** O tehnică utilă pentru optimizarea funcțiilor recursive costisitoare, implementată fie prin obiecte (cache manual) fie prin closure (cum am procedat noi). Aceasta combină ideea de structuri de date (a stoca rezultate) cu programarea funcțională (funcții pure ce pot fi optimizate memorând outputul pentru inputuri).
- **Asincronism vs. Sincronism:** Am discutat despre modelul de execuție al Node.js și de ce operațiile I/O sunt asincrone.
- Cu **stream-uri** am exemplificat citirea incrementală a datelor, evitând blocajele și permițând procesarea de fișiere mari cu memorie și timp eficiente.
- Cu exemplul de **preparare a cafelei**, am ilustrat cum *ordinea logică* a pașilor se poate conserva în cod asincron, de la cele mai simple callback-uri înlănțuite până la sintaxa elegantă `async/await`. Am evidențiat beneficiile promise-urilor în evitarea *callback hell*.
- Am comparat citirea **sincronă** și **asincronă** a unui fișier: varianta sincronă simplifică codul dar blochează firul de execuție, pe când cea asincronă menține aplicația reactivă. Am arătat și cât de ușor devine codul asincron de citire a fișierelor folosind `await`, combinând claritatea cu eficiența.
- **Tratamentul erorilor:** În OOP și cod sincron, folosim adesea `try...catch` și tipuri de erori (`Error`, `TypeError` etc.) pentru a semnala condiții de eroare. În cod asincron, când folosim

callback-uri, trebuie să verificăm erorile manual (convenția `err` ca prim argument). Cu promise-uri/async, putem reveni la `try...catch` în blocuri `async` sau la metoda `.catch` pe promise-uri. Am învățat importanța propagării și gestionării corecte a erorilor, mai ales în context asincron unde, dacă nu sunt prinse, pot trece neobservate sau chiar opri întreg procesul Node (în cazul excepțiilor ne-prinse).

În concluzie, **JavaScript** ne oferă instrumente puternice pentru a modela obiecte și relații între ele, similar cu alte limbaje orientate pe obiecte, dar cu particularitatea prototipurilor și cu flexibilitatea unui limbaj dinamic. Simultan, în mediul **Node.js**, trebuie să gândim în termeni asincroni: să profităm de mecanismele non-blocante pentru a realiza aplicații performante, fără a ne bloca în operații lente. Prin exercițiile și exemplele discutate, ar trebui să vă fie (ceva) mai clar cum să structurați codul OOP în JavaScript și cum să abordați problemele de flux de execuție asincronă.

Vă recomandăm să exersați conceptele în proiecte mici (de exemplu, extindeți codul de memoizare pentru alte funcții, sau scrieți un mic script care citește date JSON asincron și le procesează), deoarece prin practică aceste noțiuni se vor fixa mult mai bine.

May the *sudo* powers be always with you!