BASED ON student.nextlab.tech

FUNCȚII ȘI ARRAY-URI ÎN JAVASCRIPT
prelucrarea funcțională a datelor

1) CONTEXT

Fluxul dedicat prelucrării funcţionale a datelor pune accent pe metodele `filter`, `map` şi `reduce`, precum şi pe definirea unor funcţii de nivel înalt. Participantul învaţă astfel să scrie cod concis şi expresiv, adoptând un stil declarativ specific programării funcţionale. În JavaScript, funcţiile sunt obiecte de primă clasă și pot fi transmise ca parametri altor funcții (callback-uri). Funcțiile care primesc alte funcții drept argument sau returnează funcții sunt numite *funcții de ordin înalt* (higher-order functions)[1]. Metodele de array precum `filter`, `map` sau `reduce` sunt exemple de astfel de funcții de nivel înalt: ele *iter-ează* automat peste elementele unui array și aplică o funcție asupra fiecărui element[2]. Aceste metode permit prelucrarea *funcțională* a datelor: în locul buclelor imperative, folosim apeluri succinte ce descriu **ce** dorim să obținem (filtrare, transformare, agregare), delegând detaliile de parcurs intern acestor funcții.

Vom explora în detaliu șapte aplicații reprezentative care folosesc funcțiile și array-urile în stil funcțional, alternând între teorie și practică (cazuri reale). Fiecare problemă este rezolvată prin mai multe variante de cod (A, B, C, ...), evidențiind diferite abordări – de la cea imperativă clasică, la cea modernă, concisă, funcțională – precum și compromisul dintre lizibilitate, performanță, utilizarea memoriei și mutația datelor. Contextul asigură continuitatea didactică: ne bazăm pe cunoștințele de bază (bucle, funcții simple, concepte ES6 precum *arrow functions*, rest/spread etc.) și le extindem către paradigma funcțională.

OBIECTIVE DE ÎNVĂȚARE: La finalul parcurgerii materialului, studentul va fi capabil să:

- Aplice metodele `filter`, `map` și `reduce` pentru a procesa array-uri, înțelegând modul lor de funcționare și avantajele față de buclele tradiționale.
- Definieze și utilizeze **funcții variadice** (cu număr variabil de parametri) folosind atât tehnicile vechi (obiectul `arguments`), cât și pe cele moderne (parametrul *rest* `...`).
- Construiască și **extindă prototipul** array-urilor implementând propriile versiuni `Array.prototype.myMap` și `Array.prototype.myReduce`, pentru a înțelege mecanismele interne ale acestor metode.
- Utilizeze **lanțuri de apeluri** (method chaining) pe șiruri de caractere și array-uri (ex: `split().map().join()`) pentru a realiza transformări complexe într-o manieră declarativă.
- Sorteze array-uri de obiecte pe baza unui anumit câmp, definind funcții de comparație personalizate și înțelegând modul de lucru al lui `Array.sort` (inclusiv faptul că modifică array-ul *in-place*).
- Combine operațiile funcționale pentru a realiza calcule agregate precum **suma sau media** elementelor, evidențiind modul în care `reduce` poate cumula valori numerice.

Precondiții și mediu: *Este recomandat ca rularea codurilor să se facă în mediul Node.js (din terminal, ex. PowerShell/CMD sau Terminal macOS) sau direct în consola unui browser web modern. Studenții ar trebui să aibă instalat Node.js și un editor de cod (precum Visual Studio Code) și să cunoască noțiuni de bază despre variabile, tipuri primare, array-uri, sintaxa funcțiilor JavaScript, operarea cu `console.log` pentru afișare și, opțional, cum se furnizează parametri din linia de comandă (`process.argv` în Node). Familiaritatea cu sintaxa ES6 (functii arrow, operatori* `...`

rest/spread) și diferențele față de sintaxa veche vor fi de ajutor pentru a înțelege pe deplin variantele moderne ale soluțiilor.

2) FUNDAMENTE TEORETICE – stil imperativ vs. funcțional, metode array esențiale

Înainte de a trece la exercițiile concrete, să recapitulăm pe scurt metodele de array și conceptele cheie:

- **`Array.filter`**: parcurge array-ul și **filtrează** elementele pe baza unui criteriu (callback boolean). Returnează un array nou conținând doar elementele pentru care funcția de test a returnat `true`[3]. Nu modifică array-ul inițial (non-mutațional)[4][5].
- **`Array.map`**: parcurge array-ul și **transformă** fiecare element prin aplicarea unei funcții de proiecție (callback de transformare). Returnează un array nou, de aceeași lungime, cu valorile transformate[6][7]. Nici `map` nu afectează array-ul original, lucrând pe o copie.
- **`Array.reduce`**: **reduce** un array la o singură valoare prin aplicarea iterativă a unei funcții de agregare (reducer). Pe rând, elementele sunt combinate (de exemplu adunate, concatenate etc.) într-un *accumulator* care este returnat la final[8]. Metoda acceptă un parametru opțional *valoare inițială* a accumulatorului. Dacă nu se specifică, primul element al array-ului va fi luat ca start și iterarea începe de la al doilea element[9][10]. Ca și celelalte, `reduce` nu modifică array-ul de intrare.
- **Funcții de ordin înalt (Higher-Order)**: `filter`, `map`, `reduce` sunt *higher-order functions*, adică funcții care primesc alte funcții ca argument (funcțiile callback pe care le furnizăm pentru a testa, transforma sau agrega)[1]. Acest mod de definire a logicii prin callback-uri conduce la un stil declarativ: codul descrie direct operația dorită (ex: "filtrează elementele care îndeplinesc condiția X") în loc să detalieze pașii de iterare. Rezultatul este un cod adesea mai **scurt, simplu și ușor de citit**[11].
- **Stil imperativ vs. funcțional**: În abordarea imperativă, folosim bucle (`for`, `while`) și modificăm variabile sau array-ul direct (*mutație in-place*). În abordarea funcțională, preferăm să **nu** modificăm datele originale, ci să producem structuri noi ca rezultat al prelucrării (principiul imutabilității), combinând funcții existente. De exemplu, pentru a filtra și apoi transforma un array, putem *înlănțui* apelurile: `arr.filter(cond).map(transform)` – obținând un array nou ca rezultat după ambele operații, fără a altera `arr`. Acest mod poate utiliza mai multă memorie (pentru array-urile intermediare), dar reduce riscul de efecte secundare și poate fi mai ușor de verificat și menținut[12]. Totodată, codul funcțional tinde să fie concis; vom vedea adesea soluții reduse la o singură linie cu arrow functions, unde claritatea trebuie cântărită față de densitatea sintactică.

În secțiunile următoare vom parcurge fiecare exercițiu, prezentând pe rând variante de rezolvare (A, B, C, ...) în ordine crescătoare a abstractizării și conciziei. Fiecare variantă va fi însoțită de explicații teoretice și practice, plus observații despre eficiență și bune practici acolo unde este cazul.

BASED ON student.nextlab.tech

3) FILTER & MAP – Filtrarea dupa vârste și transformarea datelor

Enunț: Se dă un array de vârste (numere întregi). Se cere eliminarea din listă a oricărei vârste mai mici de 18 ani (filtrarea minorilor), iar asupra celorlalte elemente (vârstele ≥ 18) se aplică o transformare. Scopul exercițiului este evidențierea principiului de **filtrare** (selecție pe bază de condiție) și de **proiecție** (transformarea fiecărui element selectat).

Să presupunem că array-ul de intrare este `const ages = [12, 17, 22, 8, 34, 18, 20];`. Dorim să obținem mai întâi numai valorile care reprezintă vârste adulte (18 și peste), apoi să transformăm fiecare astfel de vârstă – de exemplu, să o exprimăm cu un sufix `" ani"` sau să o dublăm (dacă contextul ar fi altul). Vom ilustra soluțiile posibile:

3.1. **Var 1** — *Imperativ, folosind bucle* (filtrare și mapare manuală)

În varianta de bază, fără a folosi metode funcționale predefinite, putem rezolva problema cu o abordare imperativă clasică:

```
// Var 1: Filter & Map imperativ (folosind bucle)
function adultsPlusTransform(agesArray) {
    const result = [];
    for (let i = 0; i < agesArray.length; i++) {
        const age = agesArray[i];
        if (age >= 18) {                // condiție de filtrare (vârsta adultă)
            const transformed = age + " ani";  // proiecție: adaugă sufixul " ani"
            result.push(transformed);    // adaugă la rezultatul filtrat+transformat
        }
    }
    return result;
}

// Exemplu de utilizare:
console.log(adultsPlusTransform([12, 17, 22, 8, 34, 18, 20]));
// Output așteptat: ["22 ani", "34 ani", "18 ani", "20 ani"]
```

Explicație: Funcția `adultsPlusTransform` parcurge manual lista de vârste cu o buclă `for`. Folosim o variabilă acumulatoare `result` în care *împingem* (`push`) doar acele vârste care trec testul `age >= 18`. În același loc, aplicăm și transformarea dorită – aici, convertim numărul la șir și concatenăm textul `" ani"`. La final, `result` conține elementele filtrate și modificate. Această soluție evidențiază pașii algoritmici clar, însă conține *cod ceremonial* (trebuie să scriem explicit structurile de control). În plus, logica de filtrare și cea de transformare sunt împletite în același parcurs.

Complexitate: O(n) în timp (trecem o dată prin array) și O(k) spațiu adițional, unde k este numărul de elemente care satisfac condiția (pentru stocarea rezultatului filtrat). Array-ul original este parcurs secvențial și nu este modificat. Stilul este imperativ și mutațional doar asupra array-ului rezultat (construit treptat).

3.2. **Var 2** — *Folosing Array.filter* (filtrare declarativă, apoi transformare imperativă)

O îmbunătățire este să separăm clar operațiile: mai întâi folosim metoda dedicată `filter` pentru a obține **doar adulții**, apoi aplicăm transformarea într-o buclă separată sau prin alt mecanism:

```
// Var 2: Combină Array.filter cu buclă pentru transformare
function adultsPlusTransform(agesArray) {
    const adults = agesArray.filter(function(age) {
        return age >= 18;    // păstrează doar vârstele >= 18
    });
    // acum 'adults' conține doar valorile 22, 34, 18, 20 (din exemplu)
    for (let i = 0; i < adults.length; i++) {
```

```
        adults[i] = adults[i] + " ani";  // aplică transformarea pe loc
    }
    return adults;
}
console.log(adultsPlusTransform([12, 17, 22, 8, 34, 18, 20]));
// Output: ["22 ani", "34 ani", "18 ani", "20 ani"]
```

Explicație: Metoda `filter` primește o funcție callback ce este apelată automat pentru fiecare element al array-ului inițial[13]. Această funcție trebuie să returneze `true` sau `false` pentru fiecare element, indicând dacă elementul să fie inclus în noul array. În cazul nostru, callback-ul este `function(age) { return age >= 18; }`, deci `filter` va construi un nou array `adults` conținând doar valorile pentru care condiția este adevărată[3]. După obținerea listei de adulți, parcurgem acest array filtrat și modificăm fiecare element prin adăugarea sufixului dorit.

Observație: *Deși aici am folosit o buclă `for` pentru transformare, puteam la fel de bine să folosim metoda `Array.map` pentru a aplica transformarea, ceea ce ne conduce la varianta următoare. În această variantă însă, am ilustrat utilizarea lui `filter` pentru a evidenția separarea procesului în doi pași clari: întâi filtrare, apoi proiecție. Array-ul original `agesArray` rămâne nemodificat, iar `adults` este o copie filtrată, pe care apoi o modificăm in-place (sau o putem transforma într-un nou array prin `map`).*

3.3. **Var 3** — *Folosing Array.filter și Array.map* (declarativ integral, fără bucle explicite)

Pentru a adopta pe deplin stilul funcțional, putem *înlănțui* direct `filter` și `map` într-o singură expresie. Astfel filtrarea și proiecția devin operații succesive pe array, fiecare returând un nou array:

```
// Var 3: Chain filter() and map() in one expression
const adultsPlusTransform = (agesArray) =>
    agesArray
      .filter(age => age >= 18)           // filtrează adulții
      .map(age => age + " ani");          // transformă fiecare în string cu " ani"

// Test
console.log(adultsPlusTransform([12, 17, 22, 8, 34, 18, 20]));
// Output: ["22 ani", "34 ani", "18 ani", "20 ani"]
```

Explicație: Aici am definit `adultsPlusTransform` ca funcție săgeată (arrow) care returnează direct rezultatul compoziției între `filter` și `map`. Callback-urile le-am scris folosind sintaxă arrow concisă: - `age => age >= 18` – funcție de filtrare ce returnează true pentru valori >= 18. - `age => age + " ani"` – funcție de mapare ce concatenează stringul `" ani"` la fiecare valoare (JavaScript convertește automat age la string când îl concatenează cu un string).

Lanțul `agesArray.filter(...).map(...)` produce întâi un array nou al adulților, apoi imediat un alt array nou cu valorile transformate. Putem observa clar principiul *proiecției*: `map` creează un array proiectat pe baza celui filtrat[14]. Codul este mult mai scurt și evită variabile intermediare explicite. De asemenea, este **nemutațional**: nicio valoare din array-ul original nu este modificată in-situ; în schimb, se generează două array-uri noi (intermediar și final). Costul suplimentar de memorie (pentru array-ul intermediar al adulților) este de dimensiune proporțională cu numărul de elemente ≥ 18 (k elemente, cum discutam), dar pentru liste relativ mici/medii avantajele clarității pot prima.

Această variantă evidențiază exact *filtrarea* și *transformarea* ca două etape distincte, fiecare realizată de o funcție dedicată, ceea ce îmbunătățește **lizibilitatea**: citind codul, aproape că putem înțelege cerința inițială în cuvinte – “filtrează vârstele adulte și adaugă « ani » la fiecare”. Acesta este unul dintre beneficiile stilului declarativ funcțional.

3.4. **Var 4** — *Map în loc de filter (tehnică alternativă)*

Este posibil să obținem același rezultat și fără a apela direct `filter`, folosind `map` și ignorând elementele necorespunzătoare. Menționăm această abordare ca pe o curiozitate sau uneori

optimizare: dacă dorim să trecem o singură dată prin array (în loc de două ori cum face `filter+map`), putem folosi `map` pentru a genera fie un element transformat, fie un element "gol" pe care apoi să-l înlăturăm. Un exemplu de truc ar fi generarea valorii `null` sau `undefined` pentru intrările care nu satisfac condiția, apoi filtrarea acelor valori falsy. Totuși, acest procedeu este mai greu de citit și nu este recomandat pentru claritate. Iată cum ar arăta:

```
// Var 4: Folosind map() pentru a filtra implicit (mai puțin lizibil)
const adultsPlusTransform = (agesArray) =>
    agesArray
      .map(age => age >= 18 ? age + " ani" : null)
      .filter(x => x);  // filtrează valorile null (falsy) rezultate

console.log(adultsPlusTransform([12, 17, 22, 8, 34, 18, 20]));
// Output: ["22 ani", "34 ani", "18 ani", "20 ani"]
```

Explicație: În `map`, folosim o expresie ternară: dacă `age >= 18`, întoarcem `age + " ani"`, altfel întoarcem `null`. Rezultatul lui `map` va fi un array de aceeași lungime cu cel inițial, unde pozițiile corespondente vârstelor sub 18 conțin valoarea `null`, iar celelalte conțin șirurile cu `" ani"`. Apoi aplicăm `filter(x => x)` – această funcție callback returnează valoarea lui `x` evaluată ca boolean. În JavaScript, `null` este evaluat ca false, iar string-urile negoale ca true, deci efectul este păstrarea doar a elementelor transformate valide, eliminând `null`-urile. Rezultatul final coincide cu a face `filter` apoi `map`.

Această tehnică arată flexibilitatea lui `map` și `filter`, dar claritatea este afectată; practic condiția de filtrare este "ascunsă" în logica de mapare. În general, e preferabil să folosiți `filter` când intenția principală e filtrarea. Complexitatea temporală rămâne O(n) (o singură trecere completă prin date, plus o a doua trecere pentru filtrarea null-urilor – deci tot două operații seriale, similar cu `filter+map` ca număr, însă executate invers). **Concluzie:** Var.3 rămâne soluția recomandată pentru concizie și claritate.

Cazuri reale: Metoda combinată `filter().map()` este foarte frecvent întâlnită în prelucrarea colecțiilor de date. De exemplu, dacă aveți o listă de obiecte reprezentând utilizatori și doriți să obțineți numele (proprietatea `name`) tuturor utilizatorilor majori, ați putea scrie: `users.filter(u => u.age >= 18).map(u => u.name)`. Aceeași logică se aplică – se obține întâi subsetul (filtrul) relevant, apoi se proiectează fiecare element în altceva (în acest caz extragem o proprietate). Acest tipar de *filtrare urmată de proiecție* este esențial în programarea cu array-uri și ne pregătește pentru următorul exercițiu, unde vom folosi și `reduce` pentru agregare.

4) **Map & Reduce** – Filtrarea numerelor dupa un criteriu și calculul sumei lor

Enunț: Se construiește un array de numere; apoi se filtrează valorile care sunt divizibile cu un anumit parametru dat și se calculează suma acestor valori folosind `reduce`. Acest exercițiu pune în practică atât filtrarea, cât și agregarea rezultatelor cu reduce, evidențiind modul de combinare a mai multor metode funcționale.

Să presupunem că vrem să generăm o listă de pătrate perfecte ale primelor n numere naturale (1^2, 2^2, 3^2, ...), după care să selectăm dintre ele doar pe cele divizibile cu, să zicem, 3, și în final să aflăm suma acelor valori. Vom demonstra diferite abordări:

4.1. **Var 1** — *Imperativ, bucle pentru generare, filtrare și sumare*

Pentru început, putem realiza toți pașii manual:

```
// Var 1: Imperativ - generează pătrate, filtrează și însumează
function sumOfSquaresDivBy(n, divisor) {
    const squares = [];
    for (let i = 1; i <= n; i++) {
```

```
        squares.push(i * i);          // construiește array-ul de pătrate (1^2, 2^2,
..., n^2)
    }
    let sum = 0;
    for (let j = 0; j < squares.length; j++) {
        if (squares[j] % divisor === 0) {  // filtrează pe criteriul divizibilității
            sum += squares[j];            // acumulează suma
        }
    }
    return sum;
}

// Exemplu: suma pătratelor ≤ 10^2 care sunt divizibile cu 3
console.log(sumOfSquaresDivBy(10, 3));
// Explicație: pătratele până la 10: [1,4,9,16,25,36,49,64,81,100];
// cele divizibile cu 3: [9, 36, 81]; suma = 126.
// Output: 126
```

Explicație: Funcția `sumOfSquaresDivBy(n, divisor)` generează mai întâi array-ul `squares` al pătratelor numerelor de la 1 la n. Apoi parcurge acest array și testează fiecare element dacă `mod divisor === 0`; dacă da, îl adună la variabila `sum`. În final, `sum` conține suma cerută. Am decuplat aici generarea datelor de procesarea lor ulterioară, pentru claritate, însă se putea face și "din mers" (în aceeași buclă să calculăm pătratul și imediat să și testăm/adicționăm). Această soluție are complexitate O(n) (o buclă de lungime n și încă o buclă de lungime n, deci formal O(2n) ~ O(n)). Consumul de memorie este O(n) pentru stocarea vectorului de pătrate, care însă nu era neapărat necesar (puteam agrega direct).

De remarcat că aici am ilustrat și generarea unui array pe baza unei formule – calcularea pătratelor. Asemenea sarcini se pot rezolva elegant cu metode funcționale precum `Array.from` sau direct cu `map` pe o secvență de indici, după cum vom vedea mai jos.

4.2. **Var 2** — *Filter + Reduce direct pe array existent*

Dacă presupunem că avem deja un array de numere și dorim doar filtrarea și suma, putem folosi direct `filter` și `reduce` fără să ne concentrăm pe modul de construire a array-ului. Spre exemplu, considerând aceeași listă de pătrate de la 1 la 10, putem face:

```
// Var 2: Aplica filter și apoi reduce pentru sumare
const squares = [1,4,9,16,25,36,49,64,81,100];
const sumDivBy3 = squares
    .filter(x => x % 3 === 0)         // păstrează doar pe cele divizibile cu 3
    .reduce((acc, val) => acc + val, 0);  // adună toate valorile filtrate, plecând de
la 0

console.log(sumDivBy3);  // Output: 126
```

Explicație: Lanțul de metode funcționează astfel: `filter` produce un nou array (în exemplu `[9, 36, 81]`), apoi `reduce` preia acel array și îl reduce la o valoare scalară – aici suma elementelor. Funcția callback dată lui `reduce` este `(acc, val) => acc + val`, adică ia acumulatorul curent și adaugă valoarea curentă, returnând noua sumă acumulată. Ca *valoare inițială* a acumulatorului am dat `0` (al doilea argument la `reduce`), astfel încât suma începe de la 0[10]. Rezultatul este 126.

Această soluție evidențiază utilizarea compusă a metodelor și este concisă, dar presupune că `squares` era deja definit. Să revenim la cazul în care trebuie să și generăm array-ul.

BASED ON student.nextlab.tech

4.3. **Var 3** — *Generare cu Array.from + filter + reduce (complet declarativ)*

JavaScript oferă modalități succinte de a genera array-uri de anumită lungime, fără bucle explicite, prin metoda statică `Array.from`. Putem genera un array de lungime n cu valori calculate pe baza indexului, apoi aplica direct `filter` și `reduce`. Pentru pătratele numerelor 1..n:

```
// Var 3: Folosind Array.from pentru generare, apoi filter+reduce
function sumOfSquaresDivBy(n, divisor) {
    return Array.from({length: n}, (_, i) => (i+1) * (i+1))  // generează [1^2, 2^2,
..., n^2]
           .filter(x => x % divisor === 0)                // filtrează după criteriu
           .reduce((acc, val) => acc + val, 0);           // însumează valorile filtrate
}
console.log(sumOfSquaresDivBy(10, 3));  // 126
```

Explicație: `Array.from({length: n}, callback)` creează un array de lungime n și apelează `callback(element, index)` pentru fiecare index de la 0 la n-1, folosind rezultatul drept valoare. Noi ignorăm primul parametru (care ar fi un eventual element existent, dar aici pornește de la un array gol de lungime n) și folosim indexul `i` pentru a calcula `(i+1) * (i+1)` – pătratul lui i+1. Astfel, obținem direct array-ul de pătrate dorit. Apoi aplicăm `filter` și `reduce` similar cu varianta anterioară. Totul este într-o singură expresie *chain-uită*. Citirea codului ne arată clar intenția: "construiește un array de la 1 la n la pătrat, filtrează-l pe criteriu, însumează elementele".

Avantajul este că evităm variabile intermediare și scriem cod declarat aproape ca o *formulă*. ***Dezavantajul*** posibil este că, pentru un începător, expresia poate părea densă; de aceea am prezentat anterior și pașii separați.

Tehnic, ***complexitatea*** este tot O(n) (trecem prin 3 etape lineare, deci ~3n operații, dar constant de același ordin). `Array.from` creează array-ul inițial de lungime n, deci necesită O(n) memorie pentru a-l stoca (asemănător cu bucla noastră care construia `squares`). Funcționalitatea este însă compactă și evităm un `for` explicit.

4.4. **Var 4** — *Utilizarea lui reduce pentru a filtra și suma într-o singură trecere*

O idee mai avansată este să folosim `reduce` nu doar pentru sumare, ci și pentru a **filtra** elementele pe parcurs, combinând ambele operații într-un singur parcurs al array-ului. Practic, în loc să generăm un array filtrat intermediar, putem verifica condiția în interiorul reducer-ului și să adunăm doar dacă este îndeplinită. Deși `reduce` este conceput pentru agregare, nimic nu ne oprește să punem logică de filtrare în callback-ul său. De exemplu:

```
// Var 4: Reduce într-o singură trecere (filtrare + sumare combinate)
function sumOfSquaresDivBy(n, divisor) {
    return Array.from({length: n}, (_, i) => (i+1) * (i+1))
           .reduce((acc, val) => {
               if (val % divisor === 0) {
                   return acc + val;   // acumulează doar valorile ce trec filtrul
               }
               return acc;            // altfel, menține acumularea nemodificată
           }, 0);
}
console.log(sumOfSquaresDivBy(10, 3));  // 126
```

Explicație: Aici, `reduce` pornește de la 0 și pentru fiecare `val` din array verifică dacă `val` este divizibil cu `divisor`. Doar dacă da, adaugă `val` la acumulare. Dacă nu, întoarce `acc` neschimbat. Astfel, la final `acc` conține suma doar a valorilor dorite. Am folosit tot `Array.from` pentru generarea inițială a pătratelor; alternarea generării cu calculul în același reducer ar fi

făcut callback-ul și mai complicat (am fi putut genera pătratul *și* verifica *și* aduna în același loc, dar am preferat claritatea).

Acest mod economisește crearea unui array intermediar filtrat, parcurgând lista o singură dată. În termeni de eficiență, reduce operațiunile de iterare, dar codul devine mai puțin declarativ – îmbină două preocupări în aceeași funcție callback. Dacă lista e foarte mare, uneori e benefic să eviți crearea de structuri intermediare, însă în majoritatea cazurilor, claritatea lui `filter`+`reduce` separat (Var 3) este preferată. Totuși, acest exemplu demonstrează flexibilitatea lui `reduce`: practic, putem realiza orice prelucrare lineară cu el, chiar și simulând comportamentul altor metode (filtrare, mapare) în interiorul său[15][16].

Cazuri reale: *Combinația `filter` + `reduce` este des întâlnită, de pildă: avem un array de tranzacții și vrem suma celor care sunt de un anumit tip – aplicăm întâi `filter` pe criteriul tipului, apoi `reduce` pentru a aduna valorile tranzacțiilor. Alternativ, un truc ca mai sus (doar `reduce`) se poate folosi dacă dorim media valorilor de un anumit tip: putem număra și aduna simultan în `reduce` și la final împărți (sau, mai clar, filtra apoi calcula media). Următorul exercițiu despre funcții variadice ne va duce într-o altă direcție, dar vom reveni la `reduce` pentru medie la final, la pasul opțional 7.*

5) **Funcții variadice** și formatare de șiruri

Enunț: Să se implementeze o funcție *variadică* ce primește un șablon (template) de text cu parametri inserați și returnează un șir de caractere formatat prin înlocuirea parametrilor cu valorile furnizate. De exemplu, apelul `format("Hello {name}", {name: "Ana"})` ar trebui să returneze `"Hello Ana"`. Aici, șirul `"{name}"` din template este un placeholder ce trebuie înlocuit cu valoarea corespunzătoare cheii `name` din obiectul dat ca argument.

O funcție *variadică* este o funcție ce poate accepta un **număr variabil de argumente**[17][18]. În cazul de față, dacă extindem conceptul, funcția de formatare ar putea accepta un număr arbitrar de parametri în șablon și valorile aferente lor. Vom analiza mai multe moduri de a implementa o asemenea funcționalitate, subliniind diferențele între abordările pre-ES6 (folosind obiectul special `arguments`) și cele moderne (folosind parametrul *rest* `...` și tehnici ES6 precum *template literals* sau *string replace* cu expresii regulate).

5.1. **Var 1** — *Înlocuire simplă, parametru fix* (non-variadic)

Pentru început, putem realiza o versiune foarte simplă a problemei, nevariadică, care se ocupă doar de un singur parametru cunoscut. Nu este soluția cerută, dar e un punct de plecare pentru a înțelege ce trebuie generalizat:

```
// Var 1: Format simplu, un singur parametru "{name}" fix
function formatName(template, nameValue) {
    return template.replace("{name}", nameValue);
}
console.log(formatName("Hello {name}, welcome!", "Ana"));
// Output: "Hello Ana, welcome!"
```

Explicație: Funcția `formatName` primește un șir `template` și un singur parametru textual `nameValue`. În implementare, folosește metoda `String.prototype.replace` pentru a înlocui prima apariție a subșirului `"{name}"` cu valoarea dată. Metoda `replace` caută un șir sau o expresie regulată și substituie cu alt șir[19]. Rezultatul este șablonul cu `{name}` substituit de exemplu cu `"Ana"`.

Această variantă funcționează doar dacă șablonul conține exact placeholderul literal `{name}` o dată. Pentru alte chei sau multiple apariții, ar trebui extinsă. De asemenea, funcția nu este *cu adevărat variadică*, deoarece semnătura ei are parametri fixați (`template` și o valoare). ***Să trecem deci la generalizare***.

5.2. **Var 2** — *Funcție variadică pre-ES6 cu obiectul `arguments`*

O abordare clasică (anterioară ES6) pentru funcții cu număr variabil de argumente este utilizarea obiectului local `arguments`. Acesta conține toate argumentele transmise, accesibile prin indici numeric (`arguments[0]`, `arguments[1]`, ...) și are o proprietate `length` cu numărul lor[20]. Să definim `format` astfel încât să poată primi o mulțime de valori ce trebuie inserate în ordine în template, în locul placeholder-elor de formă `{0}`, `{1}`, etc. Adică `format("A {0} and a {1}", "cat", "dog")` → `"A cat and a dog"`.

```
// Var 2: Variadic classic, folosește `arguments` și placeholderi numerici {0}, {1}, ...
function format() {
    let template = arguments[0];        // primul argument e șablonul
    for (let i = 1; i < arguments.length; i++) {
        // înlocuiește toate aparițiile placeholderului {i-1} cu argumentul
corespunzător
        const placeholder = "{" + (i-1) + "}";
        template = template.split(placeholder).join(arguments[i]);
    }
    return template;
}
console.log(format("A {0} and a {1} fell in love.", "cat", "dog"));
// Output: "A cat and a dog fell in love."
```

Explicație: Am definit `format` fără parametri expliciți, astfel încât să putem apela cu oricâte argumente. `arguments[0]` este considerat șablonul, iar restul (`arguments[1]`, `arguments[2]`, ...) sunt valorile de inserat. Pentru fiecare astfel de valoare, construim placeholder-ul corespunzător (de exemplu, pentru `i=1` placeholderul e `{0}`, pentru `i=2` e `{1}`, ș.a.m.d.) și apoi folosim o tehnică simplă de înlocuire: *spargem* șablonul după acel placeholder și *reunim* segmentele la loc cu valoarea intercalată.

Practic, `template.split(placeholder).join(arguments[i])` înlocuiește **toate** aparițiile placeholderului cu textul dorit (un truc util: `split().join()` echivalează cu un `replace` global pentru un substring). După ce iterăm peste toți parametrii, `template` conține textul formatat final.

Această funcție este **variadică** – putem trimite oricâte argumente suplimentare, iar ea le va folosi în ordine. Are însă limitări: - Placeholderii sunt numeric indexați `{0}`, `{1}`, ceea ce impune ca în șablon ordinea și numărul lor să se potrivească exact cu argumentele primite, altfel textul final va rămâne cu placeholderi neînlocuiți sau valorile extra vor fi ignorate. - Implementarea se bazează pe convenție (primul argument e template-ul). Dacă cineva citește semnătura `function format() { }`, nu e clar ce argumente așteaptă – e un compromis al metodei cu `arguments`. Documentația sau convenția devin esențiale.

Observație: *Am fi putut folosi și `template.replace` cu expresie regulată pentru a face substituirea globală, dar varianta cu `split().join()` este mai ușor de înțeles la prima vedere și suficientă aici.*

5.3. **Var 3** — *Funcție variadică modernă (ES6) cu parametrul rest `...args`*

ECMAScript 6 introduce posibilitatea de a capta toate argumentele suplimentare într-un array folosind sintaxa *rest*. De exemplu, `function f(...args) { }` face ca `args` să fie un array ce conține toți parametrii primiți (începând de la poziția 0)[21]. În cazul nostru, putem defini `format(template, ...values)` unde `template` este primul parametru (obligatoriu) și `...values` colectează restul argumentelor într-un array `values`. Logic, implementarea internă poate fi foarte similară cu cea de la Var 2, doar că accesăm valorile prin iterație direct pe array-ul `values`:

```
// Var 3: Variadic cu ...args (ES6), placeholderi numerici
function format(template, ...values) {
    for (let i = 0; i < values.length; i++) {
```

```
        const placeholder = `{${i}}`;
        template = template.split(placeholder).join(values[i]);
    }
    return template;
}
console.log(format("A {0} and a {1} fell in love.", "cat", "dog"));
// Output: "A cat and a dog fell in love."
```

Explicație: Diferența majoră față de Var 2 este la semnătura funcției: acum avem parametrii `(template, ...values)`. Când apelăm `format("A {0} and a {1}", "cat", "dog")`, intern: - `template` va fi stringul `"A {0} and a {1}..."`, - `values` va fi array-ul `["cat", "dog"]`.

Iterăm `i` de la 0 la `values.length-1`. Pentru fiecare, construim placeholderul exact ca înainte (`"{0}"`, `"{1}"`, ...) și înlocuim în template. Rezultatul este același. Codul e puțin mai clar, deoarece știm explicit ce reprezintă parametrii (nu mai folosim `arguments` generic). De asemenea, funcțiile arrow nu au propriul `arguments`, deci dacă am scrie `const format = (template, ...values) => { ... }`, ar trebui să folosim `values` neapărat.

Acest stil modern este de preferat, fiind mai robust și mai citibil. Totuși, încă folosim placeholderi numerici și ordinea strictă a valorilor. Să trecem la o variantă care folosește chei denumite (ca `{name}`) și un obiect de valori, care e adesea mai flexibil.

5.4. **Var 4** — *Formatare pe bază de obiect cu chei (stil Pythonic)*

În multe limbaje (ex: Python, C#) există facilități de *string interpolation* unde poți specifica nume de variabile între acolade în șablon, iar funcția de format le înlocuiește ținând cont de un context (de exemplu un dicționar de valori). Vom implementa ceva similar: `format(template, replacementsObj)` unde `replacementsObj` este un obiect ce conține perechi cheie-valoare (ex: `{name: "Ana", age: 30}`) corespunzătoare placeholderilor din text (ex: `"Salut {name}, ai {age} ani."`).

```
// Var 4: Formatare folosind obiect de înlocuire și chei numite
function format(template, replacements) {
    for (const key in replacements) {
        if (replacements.hasOwnProperty(key)) {
            const placeholder = `{${key}}`;
            template = template.split(placeholder).join(replacements[key]);
        }
    }
    return template;
}
console.log(format("Salut {name}, ai {age} ani.", { name: "Ana", age: 30 }));
// Output: "Salut Ana, ai 30 ani."
```

Explicație: Funcția primește acum doi parametri: `template` (șablonul) și `replacements` (obiectul de înlocuire). Parcurgem cheile obiectului cu un `for...in`. Pentru fiecare cheie (ex: `"name"`), construim placeholderul `"{name}"` și înlocuim toate aparițiile lui în template cu valoarea asociată (ex: `"Ana"`). Verificăm `hasOwnProperty` ca să ne asigurăm că cheia vine direct din obiect (nu din lanțul de prototipuri, deși în majoritatea cazurilor nu e o problemă aici). Rezultatul final este returnat.

Avantajul acestei abordări este că placeholderii pot apărea în ordine arbitrară sau pot fi repetați în text, atâta timp cât obiectul conține cheia respectivă. De asemenea, nu trebuie să ținem minte ordinea argumentelor; folosim nume sugestive de parametri. Asta îmbunătățește **lizibilitatea șablonului**.

Limitări: Implementarea de mai sus nu înlocuiește recursiv – adică dacă o valoare din obiect este la rândul ei un placeholder de forma `{...}`, nu îl va rezolva (dar nici nu e un requirement obișnuit). De

asemenea, dacă șablonul conține o cheie care nu e în obiect, ea va rămâne neînlocuită (poate ar merita adăugată o verificare și avertizare).

5.5. **Var 5** — *Înlocuire cu expresie regulată globală*

Metodele de mai sus folosesc `split().join()` în buclă, ceea ce este eficient pentru înlocuiri multiple, însă JavaScript ne permite și o abordare mai directă cu expresii regulate. Putem scrie o singură linie care să înlocuiască toate ocurențele pattern-ului `{cheie}` cu valoarea potrivită, folosind un callback în `String.replace`. Acesta e un nivel mai avansat, dar merită cunoscut:

```
// Var 5: Folosind regex global și replace cu callback
function format(template, replacements) {
    return template.replace(/\{([^}]+)\}/g, (_, key) => {
        return replacements[key] !== undefined ? replacements[key] : `{${key}}`;
    });
}
console.log(format("Salut {name}, ai {age} ani.", { name: "Ana", age: 30 }));
// Output: "Salut Ana, ai 30 ani."
```

Explicație: Expresia regulată `/\{([^}]+)\}/g` caută orice subșir de forma `{...}` și capturează conținutul dintre acolade (prin parantezele rotunde `(...)`). Flag-ul g asigură *căutare globală*, adică vor fi înlocuite toate aparițiile[22]. Metoda `replace` cu doi parametri – un regex și o funcție – va apela funcția noastră de fiecare dată când găsește o potrivire. Callback-ul primește ca argumente: mai întâi textul complet potrivit (ex: "{name}"), apoi grupurile capturate – în cazul nostru unul singur, `key`. Ignorăm celelalte argumente (index etc.). În corp, returnăm `replacements[key]` dacă există, altfel returnăm placeholderul original nealterat (asta asigură că dacă lipsește vreo cheie, nu se șterge placeholderul din text).

Astfel, într-o singură trecere prin text, se fac toate înlocuirile necesare. Această variantă este foarte concisă și puternică. Practic, realizează același lucru ca Var 4, dar într-un mod declarativ: "înlocuiește pattern-ul {cheie} cu ceva". E de remarcat aici sintaxa `(_, key) => ...` unde folosim `_` ca parametru pentru valoarea completă potrivită, pe care nu o folosim. Doar `key` (capturat de `([^}]+)`) ne interesează, care va fi "name" la prima apariție, "age" la a doua etc.

Complexitate: `replace` cu regex global parcurge textul o dată, deci eficiența e foarte bună. Totuși, citirea și scrierea unei expresii regulate poate fi mai dificilă pentru un începător. ***Este important de știut că pentru aceasta există alte soluții (dar nu var 5!).***

5.6. **Var 6** — *Templating modern cu Template Literals (interpolare nativă)*

Pentru completitudine, menționăm că JavaScript modern (ES6) are conceptul de *template literals* cu sintaxa backtick (`) în care putem insera expresii JS între `${...}` direct în string. De exemplu:

```
let name = "Ana", age = 30;
console.log(`Salut ${name}, ai ${age} ani.`);
// Output: Salut Ana, ai 30 ani.
```

Acest mecanism însă funcționează la momentul creării stringului în cod, nu la runtime pe un șablon arbitrar. De aceea nu îl putem folosi pentru a formata dinamic pe baza unui șablon input (decât dacă am recurge la `eval`, ceea ce nu e recomandat). Totuși, în practică, soluțiile prezentate la Var 4 sau Var 5 sunt suficiente pentru a implementa un mic utilitar de formatare.

Cazuri reale: O funcționalitate de tipul celei implementate mai sus se regăsește în multe librării sau utilitare (de ex. în Node.js, modulul `util.format` face ceva similar pentru substituirea `%s`, `%d` etc.). În dezvoltarea web, este util când vrei să generezi mesaje personalizate, șabloane de email, sau chiar pentru localizare (înlocuind placeholderi cu texte traduse). Prin exersarea variantelor de mai sus, veți

înțelege mai bine mecanismele limbajului: obiectul `arguments` vs. rest operator, manipularea stringurilor cu `replace` sau `split/join`, iterarea proprietăților unui obiect, expresii regulate etc.

6) **Implementarea funcțiilor globale** `map` și `reduce` (Polyfill)

Enunț: Se cere crearea propriilor versiuni `Array.prototype.myMap` şi `Array.prototype.myReduce` pentru a înţelege cum funcţionează aceste metode la nivel intern[23][24]. Practic, vom adăuga metode personalizate la prototipul array-ului, care să poată fi apelate pe orice array și să reproducă comportamentul lui `map` și `reduce`. Acest exercițiu ajută la consolidarea înțelegerii *callback-urilor*, a contextului `this` și a modului în care aceste metode interacționează cu array-ul.

6.1. **Var 1** — *myMap simplu*

Implementarea lui `map` presupune parcurgerea fiecărui element al array-ului, aplicarea funcției de transformare (callback-ul) și colectarea rezultatelor într-un array nou, pe care îl returnăm. Să definim `myMap`:

```
// Adăugăm myMap în prototipul Array
Array.prototype.myMap = function(callbackFn) {
    const result = [];
    for (let i = 0; i < this.length; i++) {
        if (i in this) {  // verifică existența elementului (pentru a sări peste "hole"-
uri din array-uri sparse)
            result.push( callbackFn(this[i], i, this) );
        }
    }
    return result;
};

// Exemplu:
const nums = [1, 2, 3, 4];
console.log( nums.myMap(x => x * 2) );  // [2, 4, 6, 8]
```

Explicație: Am folosit sintaxa `Array.prototype.myMap = function(...) { ... }` pentru a atașa funcția la toate array-urile. În interior, `this` se referă la array-ul concret pe care se apelează `myMap`. Creăm un array `result` gol, iterăm de la 0 la `this.length-1` și, pentru fiecare index existent, apelăm `callbackFn` cu trei argumente standard: valoarea curentă `this[i]`, indexul `i` și întregul array `this`[25]. Ce returnează `callbackFn` adăugăm (`push`) la `result`. La final, returnăm `result`. Observăm verificarea `if (i in this)`: aceasta asigură că dacă array-ul original are vreo poziție goală (t. num. *sparse array*, unde `length` e mai mare dar unele indexi nu au fost definiți explicit), nu vom invoca callback-ul pe element inexistent. Același comportament îl are și `Array.map` nativ (care sare peste elementele neinițializate).

Astfel, `myMap` ar trebui să se comporte identic cu `Array.map` în majoritatea cazurilor. De exemplu, dacă `nums` conține `[1,2,3]`, exemplul de mai sus dublează fiecare element și obține `[2,4,6]`, la fel ca `nums.map(x => x*2)` ar face.

6.2. **Var 2** — *myReduce simplu*

Implementarea lui `reduce` e puțin mai complexă, deoarece trebuie să gestionăm și valoarea inițială opțională și să decidem de la ce element începem acumularea. Iată o variantă simplificată:

```
Array.prototype.myReduce = function(callbackFn, initialValue) {
    let accumulator;
    let startIndex;
    if (initialValue !== undefined) {
        accumulator = initialValue;
        startIndex = 0;
    } else {
```

```
        // fără valoare inițială: folosim primul element ca accumulator initial
        if (this.length === 0) {
            throw new TypeError("Reduce of empty array with no initial value");
        }
        accumulator = this[0];
        startIndex = 1;
    }
    for (let i = startIndex; i < this.length; i++) {
        if (i in this) {  // similar, sărim peste goluri neinitialize
            accumulator = callbackFn(accumulator, this[i], i, this);
        }
    }
    return accumulator;
};

// Exemplu:
const nums = [1, 2, 3, 4];
console.log( nums.myReduce((sum, x) => sum + x, 0) );    // 10 (suma cu initial=0)
console.log( nums.myReduce((prod, x) => prod * x) );     // 24 (produs, fără initial -
ia 1*2*3*4)
```

Explicație: Metoda `myReduce` pe prototip primește doi parametri: `callbackFn` (obligatoriu) și `initialValue` (opțional).

În interior:

- Dacă se oferă `initialValue` (verificăm `!== undefined`), atunci setăm `accumulator = initialValue` și vom începe iterarea de la indexul 0 al array-ului.
- Dacă nu se oferă, atunci trebuie să folosim **primul element valid** al array-ului ca valoare inițială a acumulării. Conform specificației oficiale, dacă array-ul e gol și nu e dat initialValue, aruncăm o eroare de tip TypeError (pentru că nu există cu ce începe reducerea)[26][9]. Implementăm acest comportament: dacă `this.length === 0`, aruncăm eroarea. Altfel, setăm `accumulator = this[0]` și vom începe iterarea de la `index 1` (al doilea element), deoarece primul l-am folosit deja.
- Apoi, parcurgem array-ul de la `startIndex` până la sfârșit. La fiecare element existent, apelăm `callbackFn(accumulator, this[i], i, this)` și stocăm rezultatul în `accumulator`. Astfel, acumulatorul se **actualizează succesiv** cu valoarea returnată de callback[27].
- După buclă, returnăm `accumulator`, care reprezintă valoarea agregată finală.

Exemplele validate:

- Suma elementelor cu initialValue = 0: `myReduce((sum,x) => sum+x, 0)` parcurge toate elementele, acumulând suma (0 + 1 + 2 + 3 + 4 = 10).
- Produsul elementelor fără initialValue: `myReduce((prod,x) => prod*x)` ia initial accumulator = `nums[0]` adică 1, apoi înmulțește cu 2, apoi cu 3, apoi cu 4, rezultând 24.

Am inclus și comportamentul standard de a arunca eroare dacă reduce este chemat pe array gol fără initialValue – acest aspect testează înțelegerea cerințelor metodei originale.

6.3. **Var 3** — *Testare și utilizare a noilor metode*

Pentru a ne asigura că implementările funcționează corect, putem face câteva teste suplimentare, inclusiv pe array-uri sparse sau cu elemente de diferite tipuri:

```
// Array sparse (lungime 5, dar elemente doar la index 1 și 3)
const sparse = [];
sparse[1] = 10;
sparse[3] = 20;
console.log( sparse.myMap(x => x || 0) );
// Rez. asteptat: [ <1 empty item>, 10, <1 empty item>, 20, <1 empty item> ]
// (myMap ar trebui să păstreze golurile la fel ca map original).
console.log( sparse.myReduce((acc, x) => acc + (x||0), 0) );
// Asteptari: 30 (10+20, ignorând indecșii fără valoare).
```

Atât `myMap` cât și `myReduce` ar trebui să se comporte similar metodelor native în aceste situații. Prin acest exercițiu de *polyfill*, am înțeles mai bine ce fac intern aceste metode – anume, iterarea, apelul callback-ului cu parametri specifici și precauțiile legate de elemente neexistente sau valori inițiale.

Bune practici: *În mod normal, nu este recomandat să modificăm prototipul obiectelor native JavaScript, deoarece poate cauza conflicte; acest exercițiu este didactic. De asemenea, dacă am vrea ca metodele noastre să fie și mai robuste, am putea verifica tipul `callbackFn` și existența lui (aruncând eroare dacă nu e funcție) sau chiar folosi `thisArg` (al doilea parametru opțional la map/reduce care setează valoarea lui `this` în interiorul callback-ului). Am simplificat neincluzând `thisArg`, dar rețineți că metodele native îl suportă.*

7) **Cenzurare și lanțuri de apeluri** – filtrarea cuvintelor interzise

Enunț: Se dă un text (șir de caractere) și o listă de cuvinte interzise. Să se *cenzureze* toate aparițiile acelor cuvinte din text, înlocuindu-le cu aceeași lungime de caractere * (asteriscuri), folosind lanțuri succesive de metode (`split`, `map`, `join`). Practic, trebuie să transformăm textul astfel încât cuvintele indezirabile să apară mascat.

Acesta este un exercițiu clasic de prelucrare a șirurilor și pune accentul pe **method chaining** – vom folosi mai multe metode una după alta, aplicate pe rezultatul celei anterioare. De asemenea, vom vedea două abordări: una folosind prelucrări pe array (prin `split` și `map`), alta folosind direct metode pe string (de exemplu `String.replace` cu regex global, similar cu ce am făcut la formatare).

Pentru exemplificare, să considerăm textul: `"Ana are mere și pere."` și lista de cuvinte interzise `["mere", "pere"]`. Rezultatul cenzurat ar trebui să fie: `"Ana are **** și ****."` (presupunând că dorim să păstrăm lungimea fiecărui cuvânt cenzurat). Vom realiza acest lucru pas cu pas.

7.1. **Var 1** — *Imperativ, folosind `split` și bucle*

Mai întâi, o variantă de bază fără method chaining, pentru a înțelege mecanica:

```
// Var 1: Cenzurare imperativă cu split și buclă
function censorText(text, bannedWords) {
    let words = text.split(" ");  // împarte textul în cuvinte separate de spațiu
    for (let i = 0; i < words.length; i++) {
        let cleanWord = words[i].replace(/[.,!?]/g, "");  // elimină semne de punctuație
finale pentru comparare
        if (bannedWords.includes(cleanWord)) {
            // înlocuiește tot cu '*', menținând eventual semnele de punctuație la
sfârșit
            let censoredCore = "*".repeat(cleanWord.length);
            // Dacă cuvântul original avea punctuație (de ex "mere."), o adăugăm la
final
            let suffix = words[i].slice(cleanWord.length);
            words[i] = censoredCore + suffix;
        }
    }
    return words.join(" ");
}
```

```
console.log( censorText("Ana are mere și pere.", ["mere", "pere"]) );
// Output: "Ana are **** și ****."
```

Explicație: Funcția `censorText` primește un text și un array `bannedWords`. Întâi folosește `text.split(" ")` pentru a obține un array de cuvinte despărțite de spațiu. Observație: această metodă simplă va include și semnele de punctuație în cuvinte (de ex. "mere." va fi un element în listă, cu punct inclus). Pentru a corecta compararea, am curățat fiecare cuvânt de caractere non-literale folosind `replace(/[.,!?]/g, "")` – o expresie regulată ce elimină punct, virgulă, semnul exclamării sau întrebării (lista se poate extinde după nevoie). Apoi verificăm dacă varianta curățată a cuvântului se află în lista de interzise (`includes`).

Dacă da, construim un șir de asteriscuri de aceeași lungime (`"*".repeat(length)` produce un string cu `length` de asteriscuri[28]). Apoi recompunem cuvântul cenzurat: dacă cuvântul original avea semne de punctuație la final (stocate în `suffix`), le adăugăm înapoi după asteriscuri, ca să nu eliminăm punctuația din text. Înlocuim cuvântul în array cu versiunea cenzurată. La final, `words.join(" ")` reconstruiește textul, reunind toate cuvintele separate prin spațiu.

Această abordare funcționează și arată clar intenția, dar am folosit bucle imperative. Să vedem cum o transformăm într-un ***lanț de metode***.

7.2. **Var 2** — *Method chaining: `split` → `map` → `join`*

Putem realiza aceeași prelucrare într-o manieră mai declarativă utilizând lanțuri de apeluri:

```
// Var 2: Cenzurare cu split-map-join (lanț de metode)
function censorText(text, bannedWords) {
    return text
      .split(" ")
      .map(word => {
          let cleanWord = word.replace(/[.,!?]/g, "");
          if (bannedWords.includes(cleanWord)) {
              let censoredCore = "*".repeat(cleanWord.length);
              let suffix = word.slice(cleanWord.length);
              return censoredCore + suffix;
          }
          return word;
      })
      .join(" ");
}
console.log( censorText("Ana are mere și pere.", ["mere", "pere"]) );
// Output: "Ana are **** și ****."
```

Explicație: Operația este aproape identică cu cea din Var 1, doar că aici folosim `map` în loc de bucla `for`. Procesul: - `text.split(" ")` – împarte textul inițial în array de cuvinte. - `.map(word => { ... })` – transformă fiecare cuvânt după logica: - curățăm `word` de punctuație (exact ca înainte), - verificăm lista `bannedWords` (exact ca înainte), - dacă trebuie cenzurat, construim `censoredCore` de `*` și adăugăm sufixul (exact ca înainte), - returnăm fie cuvântul nemodificat (dacă nu e interzis), fie cenzurat (dacă era interzis). - `.join(" ")` – recompune șirul final din array-ul de cuvinte (unele modificate).

Acest lanț este un exemplu de *pipeline* de prelucrare: output-ul lui `split` devine input pentru `map`, iar output-ul lui `map` devine input pentru `join`. Putem citi codul ca pe o propoziție: “ia textul, împarte-l în cuvinte, înlocuiește fiecare cuvânt interzis cu asteriscuri, apoi lipește-l la loc într-o frază”. Este un exemplu clar de method chaining pe care JavaScript îl permite datorită faptului că multe metode ale obiectelor native (inclusiv `filter`, `map`, `join`) returnează un obiect (array sau string) pe care se poate aplica următoarea metodă.

BASED ON student.nextlab.tech

În această variantă, accentul e pus pe utilizarea lui `map` pentru transformarea elementelor array-ului. `split` și `join` sunt folosite doar pentru conversii string<->array. O altă abordare ar fi fost să aplicăm direct un înlocuitor global pe string, pe fiecare cuvânt interzis, eventual cu ajutorul regex (similar cu ce am făcut la formatare). Să explorăm și asta.

7.3. **Var 3** — *Cenzurare folosind `String.replace` cu regex*

O soluție alternativă, mai concisă, este să folosim `String.replace` cu un regex care să caute toate cuvintele interzise. Putem construi un pattern regex din lista de cuvinte. Un aspect tricky este să evităm să înlocuim parțial cuvinte (ex: dacă "mere" e interzis, să nu cenzurăm cuvântul "merem" – dar contextul de obicei e delimitarea prin spații sau semne de punctuație). Putem folosi `\b` (boundary) în regex pentru a marca limite de cuvânt. Exemplu:

```
// Var 3: Cenzurare cu regex global
function censorTextRegex(text, bannedWords) {
    const pattern = new RegExp(`\\b(${bannedWords.join("|")})\\b`, "gi");
    return text.replace(pattern, match => "*".repeat(match.length));
}
console.log( censorTextRegex("Ana are mere și pere.", ["mere", "pere"]) );
// Output: "Ana are **** și ****."
```

Explicație: Am creat o expresie regulată dinamică: `(${bannedWords.join("|")})` va forma un grup cu toate cuvintele interzise separate prin `|` (sau logic). De exemplu, dacă bannedWords = ["mere","pere"], pattern-ul devine `\b(mere|pere)\b`. `\b` asigură că potrivirea se face pe granițe de cuvinte (adică înainte și după match nu sunt caractere alfanumerice, ceea ce acoperă delimitarea de regulă). Flag-urile "gi" fac căutarea globală și case-insensitive (dacă dorim să ignorăm majuscule/minuscule; se poate scoate "i" dacă nu e cazul).

Funcția de replacement dată ca al doilea argument primește pentru fiecare potrivire `match` (care va fi exact cuvântul interzis găsit, ex: "mere" sau "pere" cu exact literele cum apar în text) și returnează un șir de asteriscuri de aceeași lungime. `match.length` ne asigură că păstrăm lungimea originală, indiferent de cuvânt.

Rezultatul este obținut într-o singură linie, fără a mai sparge textul în cuvinte manual. Totuși, în spate `replace` cu regex intern face un fel de iterație. Complexitatea este O(n * m) unde n e lungimea textului și m numărul de cuvinte interzise (combinate în regex). În practică, pentru liste mici de cuvinte și texte rezonabile, e eficient.

Această variantă arată puterea expresiilor regulate, dar ***e mai puțin accesibilă pentru cititorii neinițiați***. În schimb, lanțul `split-map-join` este foarte explicit și didactic, motiv pentru care a fost solicitat. În dezvoltarea reală, am putea folosi oricare din abordări în funcție de situație.

Cazuri reale: *Cenzurarea de cuvinte este folosită în aplicații de chat, forumuri, moderarea conținutului generat de utilizatori etc. Soluțiile pot deveni mult mai complexe (de exemplu folosind liste de cuvinte indezirabile și înlocuind partial, ex: transformând litere din interiorul cuvântului în asteriscuri pentru a păstra prima și ultima literă vizibile, etc.). Dar esența rămâne identificarea substrigurilor țintă și substituirea lor, lucru pe care l-am realizat fie procedural, fie declarativ cu metode funcționale.*

Acest exemplu a scos în evidență încă o dată **metoda `map`** și modul în care lanțurile de apeluri pot produce un cod clar și succint. În plus, am recapitulat folosirea lui `split` și `join` pentru trecerea de la text la array de cuvinte și invers, ceea ce e un pattern util de reținut.

BASED ON student.nextlab.tech

8) **Sortare** – Compararea funcțiilor în `Array.sort`

Enunț: Se sortează un array de obiecte după un anumit câmp folosind `Array.sort` şi funcţii de comparaţie. Se evidenţiază modul de definire a funcțiilor de comparație și faptul că `sort` apelează această funcție pentru a decide ordinea elementelor.

Metoda `Array.prototype.sort(compareFn)` poate sorta elementele unui array conform unui criteriu personalizat dat de `compareFn`. Această funcție de comparație primește de obicei două elemente și returnează un număr negativ, zero sau pozitiv, indicând ordinea lor[29][30]. Vom exemplifica sortarea unui array de obiecte (de exemplu, o listă de persoane cu un atribut `age`), după acel atribut numeric, atât crescător cât și descrescător, precum și sortarea lexicografică după un șir (ex: nume).

8.1. Var 1 — *Sortare numerică cu comparator simplu*

Considerăm un array:

```
const persons = [
    { name: "Andrei", age: 25 },
    { name: "Maria", age: 30 },
    { name: "Ion", age: 20 },
    { name: "Elena", age: 30 }
];
```

Dacă aplicăm direct `persons.sort()`, JavaScript va încerca să convertească elementele (obiecte) la string și să le compare după acele stringuri – care vor fi `"[object Object]"` pentru toți, deci sortarea n-ar fi utilă. Trebuie să îi dăm un `compareFn` care să compare cum dorim. Pentru sortare după vârstă crescător:

```
// Var 1: sortare după age (crescător)
persons.sort((a, b) => a.age - b.age);
console.log(persons);
```

Output (exemplificativ, ordinea lui Andrei vs Elena depinde de stabilitate):

```
[
  { name: "Ion", age: 20 },
  { name: "Andrei", age: 25 },
  { name: "Maria", age: 30 },
  { name: "Elena", age: 30 }
]
```

Explicație: Am trecut la `sort` o funcție arrow `(a, b) => a.age - b.age`. Aceasta ia două obiecte `a` și `b` și calculează diferența dintre atributele lor `age`. Conform convenției, dacă întoarce un număr negativ, `a` va fi considerat mai mic deci plasat înaintea lui `b`; dacă întoarce pozitiv, `a` după `b`; dacă întoarce 0, sunt egale din punct de vedere al sortării[31]. Operația `a.age - b.age` exact asta face: de ex, pentru `a.age=25, b.age=30` rezultă -5, deci `a` (25) vine înainte de `b` (30). Pentru două elemente ambele cu `age=30` (Maria și Elena), rezultă 0, deci sortarea le lasă în ordinea inițială. **Notă:** ECMAScript garantează acum stabilitatea sortării (din 2019)[32][33], deci Maria și Elena își păstrează ordinea originală relativă (Maria era înainte, rămâne înainte).

Dacă dorim ***sortare descrescătoare***, putem inversa comparația sau pur și simplu schimba diferența:

```
persons.sort((a, b) => b.age - a.age);  // descrescător după age
```

Acum ordinea ar deveni 30,30,25,20 (Maria, Elena, Andrei, Ion).

BASED ON student.nextlab.tech

8.2. **Var 2** — *Sortare lexicografică (după nume)*

Pentru sortarea după un șir de caractere (ex: proprietatea `name`), compararea prin scădere nu are sens (scăderea se aplică numeric). Trebuie să definim comparatorul astfel:

```
// Var 2: sortare după nume (alfabetic, case-insensitive)
persons.sort((a, b) => {
    const nameA = a.name.toLowerCase();
    const nameB = b.name.toLowerCase();
    if (nameA < nameB) return -1;
    if (nameA > nameB) return 1;
    return 0;
});
console.log(persons);
```

Presupunând lista inițială, output-ul va fi ordonat alfabetic după nume: Andrei, Elena, Ion, Maria.

Explicație: În comparator, am convertit ambele nume la lowercase pentru a face compararea case-insensitive (ca "Maria" vs "andrei" să se compare corect). Apoi: - Dacă `nameA` este lexical mai mic (vine înainte în alfabet) decât `nameB`, returnăm -1 (a < b deci a înainte). - Dacă `nameA` e mai mare lexicografic, returnăm 1. - Dacă sunt egale (după ignorarea cazului literelor), returnăm 0.

Această structură if/else echivalează cu folosirea `localeCompare` sau alte trucuri, dar e clară. Putem folosi și:

```
persons.sort((a, b) => a.name.localeCompare(b.name));
```

care sortează direct după ordine lexicografică locală (dar asta ține cont de “default case”, de aceea în exemplul nostru am folosit toLowerCase pentru consistență)[34].

8.3. **Var 3** — *Observații despre mutabilitate și stabilitate*

Este important de subliniat că `sort` **modifică array-ul original in-place**[35][36]. În exemplul de mai sus, `persons` este reorganizat, nu s-a returnat o copie sortată (metoda însăși returnează tot array-ul sortat, dar referința e aceeași). Dacă dorim să păstrăm array-ul original, ar trebui să lucrăm pe o copie (ex: `const sorted = [...persons].sort(...)` sau să folosim noua metodă ES2023 `toSorted` care nu modifică originalul[37]).

Am atins deja conceptul de *sortare stabilă*: dacă doi itemi sunt egali după criteriu, ordinea lor relativă rămâne cea inițială (în implementările moderne). În exemplul cu vârsta, Andrei (25) rămâne înaintea altora de 25 dacă ar fi existat, iar Maria înaintea Elenei (ambele 30) după sortare, cum am menționat[33].

Caz practic: *Sortarea personalizată este extrem de utilă. De pildă, aveți un array de produse cu preț și nume; puteți sorta după preț crescător cu `products.sort((p,q)=>p.price-q.price)` sau după nume cu comparatorul lexicografic. Pentru numere e comun să vedeți sintaxa scurtă `(a,b)=>a-b` (crescător) sau `(a,b)=>b-a` (descrescător)[38]. Fără comparator, `.sort()` convertește totul în string și ordonează după codul Unicode al caracterelor, ceea ce duce la rezultate neașteptate pentru numere (de exemplu, [1, 2, 10] sortat default devine [1, 10, 2] deoarece "10" < "2" ca șir)[39].*

În concluzie, `Array.sort` devine foarte puternic când îi furnizăm o funcție de comparație adecvată problemei. Trebuie doar atenție la implementarea corectă a comparatorului (să returneze valori negative, zero sau pozitive conform criteriului[40]) și la faptul că sortarea se face in-place.

BASED ON student.nextlab.tech

9) (Opțional) **Calcularea mediei** elementelor unui array cu `reduce`

Pentru a consolida utilizarea lui `reduce`, propunem ca pas opțional calcularea **mediei** elementelor unui array. Media aritmetică se poate obține sumând toate valorile și împărțind la numărul lor. Cu `reduce`, avem două posibilități: 1. Să folosim `reduce` pentru a calcula suma, apoi să împărțim la `length` în afara reducer-ului. 2. Să calculăm direct media într-o singură trecere, adunând fiecare element împărțit la `length` (sau menținând atât suma cât și count-ul în accumulator).

9.1. **Var 1** — *Suma cu reduce + împărțire exterioară*

```
const values = [10, 20, 30, 40];
const sum = values.reduce((acc, val) => acc + val, 0);
const average = sum / values.length;
console.log("Media:", average);  // 25
```

Simplu și clar: obținem `sum = 100` cu reduce, apoi `100/4 = 25`. Acest mod face două treceri (una pentru reduce, una implicată în `.length` care e O(1) de fapt). E probabil cea mai preferată variantă pentru lizibilitate.

9.2. **Var 2** — *Calcul direct în reduce (un singur parcurs)*

Putem combina operațiile într-un singur `reduce`:

```
const average = values.reduce((acc, val, index, arr) => {
    acc += val;
    if (index === arr.length - 1) {
        // la ultimul element, calculăm media
        return acc / arr.length;
    }
    return acc;
}, 0);
console.log("Media:", average);  // 25
```

Explicație: Am folosit parametrul suplimentar al callback-ului `reduce`: al patrulea argument `arr` este chiar array-ul original[41]. La fiecare pas adunăm `val` în `acc`. Când ajungem la ultimul index (`index === arr.length - 1`), în loc să returnăm suma, returnăm direct `acc / arr.length`. Astfel, reducer-ul nostru va reține ca rezultat final media. Acest artificiu face ca `reduce` să nu mai întoarcă suma (cum ar face în mod normal), ci media. E o soluție compactă, însă puțin mai dificil de urmărit: practic folosim `reduce` ca să obținem două lucruri deodată (suma și știm și numărul de elemente). O altă variantă ar fi fost să folosim `acc` ca un obiect `{ sum: ..., count: ... }` și la final să calculeze media, dar aici am profitat de parametrul `index`.

De menționat că dacă array-ul e gol, abordarea de mai sus ar trebui protejată (nu poți face media unui șir vid - ar trebui definită ca `NaN` sau aruncată o eroare).

Caz practic: *Calculul mediei cu reduce arată puterea acestuia de a produce rezultate scalare nu doar prin sumă, ci orice calcul cumulativ. În practică, să zicem că avem un array de recenzii cu rating-uri numerice și vrem rating-ul mediu – putem obține ușor suma cu reduce și apoi media. De asemenea, conceptul se extinde: am fi putut calcula dintr-un pas și deviația standard sau alte agregate, dacă ne jucăm cu accumulatorul.*

CONCLUZII

În această lecție, am parcurs o serie de probleme și soluții menite să ilustreze **utilizarea funcțiilor și array-urilor în JavaScript**, cu accent pe **prelucrarea funcțională a datelor**. Am învățat cum metode

precum `filter`, `map` și `reduce` ne permit să scriem cod declarativ, concis și expresiv, evitând adesea buclele verbose și actualizările manuale de structuri. De asemenea, am explorat conceptul de funcții variadice și tehnici de formatare a șirurilor, evidențiind tranziția de la vechiul mod cu `arguments` la sintaxa modernă cu `...rest`. Implementarea proprie a lui `map` și `reduce` ne-a dezvăluit dedesubturile acestor metode și ne-a consolidat înțelegerea callback-urilor și a prototipurilor. Prin exercițiul de cenzurare a textului am practicat *method chaining* și combinarea funcțiilor de string și array, iar prin sortare am văzut puterea funcțiilor de comparație și importanța cunoașterii comportamentului lui `sort`. În final, am demonstrat calculul mediei cu `reduce`, subliniind că această metodă nu se limitează la sume sau produse, ci poate fi adaptată la o varietate de calcule agregate.

Beneficiile abordării funcționale: Codul tindă să fie mai ușor de testat unitate cu unitate (fiecare metodă realizează o operație specifică), mai ușor de paralelizat (în anumite medii, operațiile pe colecții pot fi distribuite), și de multe ori mai scurt. Pe de altă parte, trebuie folosit cu discernământ: uneori lanțurile foarte lungi pot deveni greu de urmărit, iar crearea de array-uri intermediare inutile poate impacta performanța pentru volume foarte mari de date. Ca programatori, trebuie să echilibrăm **lizibilitatea** și **eficiența**, iar lecția de față ne-a oferit exemple de ambele perspective (imperativ vs funcțional) pentru a ne forma un simț critic în alegerea soluțiilor.

În concluzie, prin practica variatelor exemple și variante A–F/G/H propuse, studenții au ocazia să deprindă nu doar *cum* se folosește o anumită funcționalitate, ci și *de ce* o anumită abordare este mai bună într-un context. Fiecare variantă evidențiază un aspect: claritate, concizie, compatibilitate, performanță sau pur și simplu o caracteristică a limbajului JS. Acest mozaic de soluții echipează programatorul în devenire cu instrumentele necesare pentru a aborda probleme diverse într-un mod elegant și eficient, scriind cod JavaScript “la nivel înalt”.

SURSE CITATE:

- Documentația și tutoriale despre metodele de array: filter, map, reduce[42][8], sort[29][43] – concepte și exemple care au inspirat explicațiile de mai sus.
- Articole educative privind funcțiile de ordin înalt și programarea funcțională în JS[1][44] – au oferit context teoretic despre callback-uri și beneficii.
- Resurse MDN și altele pentru implementarea polyfill (myMap, myReduce)[25][27] și pentru mecanisme interne (arguments vs rest)[20], (behavior sort)[32], regex replace[19] etc., care asigură acuratețea detaliilor tehnice.

[1] [11] [15] [16] [44] Higher Order Functions: How to Use Filter, Map and Reduce for More Maintainable Code

https://www.freecodecamp.org/news/higher-order-functions-in-javascript-d9101f9cf528/

[2] [3] [6] [8] [41] [42] JavaScript Map, Reduce, and Filter - JS Array Functions Explained with Code Examples

https://www.freecodecamp.org/news/javascript-map-reduce-and-filter-explained-with-examples/

[4] [5] [7] [13] [14] JavaScript Array methods: Filter, Map, Reduce, and Sort - DEV Community

https://dev.to/ivanadokic/javascript-array-methods-filter-map-reduce-and-sort-32m5

[9] [10] [26] ecmascript 6 - Average with the Reduce Method in JavaScript - Stack Overflow

BASED ON student.nextlab.tech

https://stackoverflow.com/questions/52139703/average-with-the-reduce-method-in-javascript

[12] [17] [18] [20] [21] ___seminar2video4_explicativ_4arguments.docx

file://file_000000006b646246b13924fbba1d38da

[19] [22] [28] Program to replace a word with asterisks in a sentence - GeeksforGeeks

https://www.geeksforgeeks.org/dsa/program-censor-word-asterisks-sentence/

[23] [24] [25] [27] Polyfill for Array methods: map(), filter() and reduce() | by Tanya Singh | Nerd For Tech | Medium

https://medium.com/nerd-for-tech/polyfill-for-array-map-filter-and-reduce-e3e637e0d73b

[29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [43] Array.prototype.sort() - JavaScript | MDN

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort