FUNCȚII ȘI ARRAY-URI

TEORIE: filter, map, reduce și altele

Introducere

În acest curs vom explora modul în care funcțiile și array-urile pot fi folosite împreună pentru a scrie cod concis și expresiv, urmând un **flux de prelucrare funcțională a datelor**. Ne vom concentra pe metodele esențiale ale obiectului Array: **`filter`** (filtrare), **`map`** (proiecție/transformare) și **`reduce`** (reducere/aglomerare), precum și pe definirea de funcții de nivel înalt (high-order functions) și alte operații utile. Vom parcurge o serie de exemple practice care ilustrează aceste concepte pas cu pas, subliniind avantajele abordării funcționale.

Ce vom învăța (concret):

- Să folosim metoda `filter` pentru a **selecta** elementele relevante dintr-un array (filtrare după condiții) și metoda `map` pentru a **transforma** elementele (proiecție în alt format) – scriind cod mai declarativ în locul buclelor clasice.
- Să aplicăm împreună `map` și `reduce` pentru a realiza **prelucrări și agregări** pe array-uri (de ex. filtrarea valorilor după criterii și calculul sumei sau al altor operații cumulativ).
- Să definim **funcții variadice** (engl. *variadic functions*) – funcții care acceptă un **număr variabil de parametri** – și să implementăm un exemplu practic de formatare a șirurilor de caractere folosind parametri dinamici.
- Să înțelegem implementarea internă a metodelor `map` și `reduce` prin crearea propriilor versiuni `Array.prototype.myMap` și `Array.prototype.myReduce`, consolidând astfel cunoștințele despre aceste mecanisme de bază și despre conceptul de funcții de nivel înalt (funcții care primesc drept parametri sau returnează alte funcții)[1][2].
- Să utilizăm **lanțuri de apeluri** (method chaining) pe array-uri și string-uri (ex.: `split().map().join()`) pentru a realiza operații complexe (precum cenzurarea anumitor cuvinte într-un text) într-un stil fluent și compact.
- Să ordonăm elementele unui array de obiecte folosind metoda `sort` cu o **funcție de comparație** personalizată, evidențiind modul în care funcțiile pot fi transmise ca parametri (comparatori) și influențează comportamentul algoritmului de sortare.
- (Opțional) Să combinăm tehnica de `reduce` cu operații aritmetice pentru a calcula valori agregate mai complexe, de exemplu **media** elementelor unui array, într-un mod elegant.

Fiecare secțiune de mai jos corespunde unui **pas** în acest parcurs și conține exemple de cod (în JavaScript), explicații în limbaj natural, pseudocod unde este cazul și mențiuni despre posibile provocări sau aplicații în contexte de volum mare de date (Big Data). La final, veți găsi un scurt **quiz** explicat, pentru a verifica și fixa cunoștințele dobândite. Să începem!

PASUL 1 – Filtrare și Mapare (Filter & Map)

Concept și exemplu: Să presupunem că avem un array de vârste (numere întregi) și dorim să eliminăm toate valorile mai mici de 18 (filtrare) și apoi să aplicăm o transformare asupra vârstelor rămase (mapare). Acest exemplu ilustrează **principiul filtrării și al proiecției** – extragem elementele care ne interesează și le proiectăm/transformăm într-o nouă formă[3][4]. În termeni simpli, `filter` *selectează un subset* din datele inițiale, iar `map` *construiește un nou set* de date prin transformarea fiecărui element selectat.

În JavaScript, metoda `filter(callback)` parcurge array-ul și *reține* doar elementele pentru care `callback` întoarce `true` (callback-ul este o funcție test ce primește elementul curent și returnează true/false). Metoda `map(callback)` parcurge array-ul și *transformă* fiecare element prin funcția `callback` dată, construind un alt array de aceeași lungime cu rezultatele. Ambele metode **nu modifică array-ul original**, ci returnează array-uri noi (acest lucru este important de reținut pentru a evita efecte neașteptate asupra datelor originale).

Exemplu practic (array de vârste):

Imaginați-vă următoarea listă de vârste, în ani, stocată într-un array JavaScript:

```
const varste = [16, 21, 15, 30, 18, 14, 25];  // exemplu de array de vârste
```

Dorim să obținem doar vârstele **de cel puțin 18 ani** (majorat) și, pe acestea, să le transformăm, de exemplu, adăugând cuvântul "ani" după valoare (pentru claritate în output). Putem realiza acest lucru combinând `filter` și `map` astfel:

```
const adulte = varste
    .filter(v => v >= 18)            // păstrează doar valorile >= 18
    .map(v => v + " ani");           // convertește fiecare valoare la string, adăugând
' ani'
console.log(adulte);
// Output așteptat: [ "21 ani", "30 ani", "18 ani", "25 ani" ]
```

În exemplul de mai sus, `filter(v => v >= 18)` va returna un array intermediar cu elementele care trec testul (adică `[21, 30, 18, 25]`, eliminând 16, 15, 14 care sunt sub 18). Apoi `map(v => v + " ani")` transformă acest array filtrat într-unul nou, unde fiecare număr a devenit un șir de caractere ce include textul " ani". Variabila `adulte` va conține rezultatul final (vârstele adulte sub formă de stringuri cu unitatea "ani").

Pseudocod ilustrativ: Pentru a înțelege procedura, iată o descriere pas cu pas în pseudocod a operației de mai sus:

```
inițializează lista rezultat ca listă goală
pentru fiecare element v din varste:
    dacă v >= 18 atunci:
        adaugă v în lista filtrată
pentru fiecare element x din lista filtrată:
    transformă x în (x + " ani") și adaugă-l la lista rezultat
returnează lista rezultat
```

Acest pseudocod corespunde exact comportamentului combinat al lui `filter` și `map`: prima buclă filtrează elementele eligibile, a doua buclă le transformă. Metodele array fac intern aceste bucle pentru noi, într-un mod mai **declarativ** și concis.

Explicații și observații: În loc să scriem două bucle `for` sau să modificăm manual array-ul, folosirea lui `filter` și `map` ne permite să exprimăm **clar intenția**: "filtrează elementele care îndeplinesc condiția și apoi mapează (transformă) acele elemente într-un nou format". Codul este mai succint și mai ușor de întreținut. Totodată, evitând mutațiile in-place, reducem riscul de bug-uri cauzate de modificarea datelor originale fără intenție.

Sfat: Țineți cont că `filter` și `map` returnează * întotdeauna * un array nou și nu modifică array-ul inițial. Dacă aveți nevoie doar să **iterați** și să efectuați o acțiune (fără să construiți un nou array), folosiți `forEach`. Dacă vreți să **transformați** sau **filtrați** datele, folosiți `map` și `filter`. De asemenea, combinația lor se poate înlănțui: în exemplul de mai sus am apelat direct `.filter(...).map(...)` în lanț. Acest lanț de apeluri face ca rezultatul lui `filter` să

fie intrarea lui `map`, simplificând scrierea. Chaining-ul funcționează deoarece `filter` produce un array (pe care putem imediat aplica `map`). Vom vedea mai multe despre lanțuri de metode la un pas ulterior.

Legătura cu Big Data: Chiar dacă exemplul de aici este simplu și se referă la un array în memorie, principiul de **filtrare** și **proiecție** se aplică și pe seturi mari de date. În contexte de tip *Big Data*, operațiile de *filter* și *map* pot fi distribuite pe mai multe noduri de calcul, fiecare prelucrând o parte din date. De exemplu, într-un sistem distribuit, am putea filtra miliarde de înregistrări pentru a reține doar pe cele relevante și apoi să transformăm fiecare înregistrare filtrată – toate aceste operații fiind efectuate în paralel pe clustere de calcul, folosind aceeași paradigmă funcțională ca în codul nostru de mai sus[5][3]. Vom reveni imediat și asupra conceptului de *MapReduce*, fundamental în procesarea volumelor mari de date.

PASUL 2 – Map și Reduce

Concept și exemplu: Dacă `filter` și `map` ne permit să extragem și să transformăm date, metoda **reduce** este cea care ne permite să **aglomerăm sau să agregăm** un întreg array la o singură valoare (sau un rezultat colectiv). Numele provine din ideea de a *reduce* multipli termeni la unul singur (gândiți-vă la suma tuturor elementelor, la calcularea produsului tuturor numerelor, la combinarea obiectelor într-un singur obiect etc.). `reduce(callback, valoareInițială)` va parcurge pe rând elementele array-ului și va acumula rezultatul într-o variabilă acumulatoare, pe care o returnează la final. Callback-ul furnizat de noi dictează **cum** se combină două valori: el primește de obicei `accumulator` (valoarea acumulată până la elementul curent) și `elementul curent` și întoarce noua valoare acumulată.

În acest pas, vom construi un array de numere, vom **filtra** valorile care sunt divizibile cu un anumit parametru (un număr dat) și apoi vom folosi `reduce` pentru a **calcula suma** acelor valori filtrate[6]. Acest exemplu arată cum putem combina mai multe operații: întâi o filtrare (sau transformare) și apoi o **agregare** a rezultatelor. În practică, astfel de lanțuri de prelucrări sunt foarte comune.

Exemplu practic (filtrare și sumă cu reduce):

```
const numere = [10, 5, 8, 30, 15, 3];   // array exemplu de numere
const k = 5;                            // parametrul de divizibilitate
const sumaDivizibile = numere
    .filter(x => x % k === 0)           // păstrează doar numerele divizibile cu k
    .reduce((acc, x) => acc + x, 0);    // calculează suma celor rămase (0 este valoarea
inițială a sumei)
console.log(sumaDivizibile);
// Output: 60  (deoarece numerele divizibile cu 5 sunt 10, 5, 30, 15, iar 10+5+30+15 =
60)
```

Să explicăm pe scurt ce se întâmplă în codul de mai sus:

- **Filtrare:** `numere.filter(x => x % k === 0)` produce un sub-array care conține doar valorile din `numere` ce sunt divizibile cu `k`. Dacă `k = 5`, filtrarea va reține `[10, 5, 30, 15]` (elementele din listă care împart pe 5 fără rest).
- **Reducere (Sumă):** Pe array-ul filtrat, aplicăm `reduce((acc, x) => acc + x, 0)`. Aici folosim o funcție anonimă `(acc, x) => acc + x` care ia acumulatul curent `acc` (inițial valoarea 0, dată ca argument) și adaugă elementul `x`. Practic, `reduce` va face: `acc=0` la start, apoi acc=0+10, acc=10+5, acc=15+30, acc=45+15, rezultând `60`. La final, `reduce` returnează suma totală. Astfel, `sumaDivizibile` va conține rezultatul agregării, o singură valoare numerică.

Observați că am folosit `0` ca valoare inițială pentru acumulare (al doilea argument la `reduce`). Dacă am fi omis acest argument, `reduce` ar fi luat implicit primul element al array-ului filtrat drept start și ar

fi început suma de la al doilea element. În cazul de față, am preferat să specificăm explicit 0 pentru claritate (și pentru a ne asigura că funcționează și dacă array-ul filtrat ar fi fost gol – situație în care, fără valoare inițială, `reduce` ar arunca o eroare).

> **Notă despre reduce:** Callback-ul primit de `reduce` poate accepta până la patru argumente în JavaScript: `accumulator`, `currentValue`, `currentIndex` și `array` (referința la array-ul original). În exemplul nostru am folosit doar pe primele două, deoarece pentru sumare indexul și array-ul complet nu erau necesare. `reduce` este foarte flexibil – prin funcția furnizată putem decide ce fel de agregare să facem (suma, maximul, concatenarea șirurilor, combinarea obiectelor etc.). La final, `reduce` întoarce o **singură valoare** (rezultatul final acumulat).

Pseudocod (suma numerelor divizibile cu k):

```
acc = 0
pentru fiecare element x din numere:
    dacă x % k == 0 atunci:
        acc = acc + x    // adună valoarea la sumă
returnează acc
```

Acest pseudocod arată procedura manuală echivalentă cu filtrarea și reducerea de mai sus. `filter` + `reduce` ne permit să exprimăm aceeași logică într-o singură linie, punând accent pe *ce* condiție vrem (x % k == 0) și *ce* dorim să facem cu valorile (să le adunăm).

Map + reduce – transformări urmate de agregare: În exemplul precedent, am combinat `filter` și `reduce`. Însă uneori dorim să aplicăm întâi un `map` (o transformare) și apoi să agregăm rezultatele cu `reduce`. De pildă, dacă am dori să calculăm **suma pătratelor** numerelor care sunt divizibile cu 5, am putea scrie:

```
const sumaPatrateDiv5 = numere
    .filter(x => x % 5 === 0)
    .map(x => x * x)                  // transformă fiecare număr în pătratul său
    .reduce((acc, x) => acc + x, 0);  // apoi adună toate pătratele
console.log(sumaPatrateDiv5);
// În exemplul nostru: numere divizibile cu 5 -> [10,5,30,15]; pătrate ->
[100,25,900,225]; sumă = 1250
```

Această abordare „filtrează apoi transformă apoi reduce” evidențiază foarte bine stilul **funcțional**: luăm o colecție de valori și aplicăm pe rând mai multe *funcții* independente pentru a obține rezultatul final. Fiecare etapă are un scop clar și evităm să scriem cod boilerplate de iterare manuală.

***NOTĂ* – legătura cu Big Data (MapReduce):** Merită menționat că **map** și **reduce** dau numele unui faimos model de programare distribuită numit chiar *MapReduce*. În sistemele de tip Big Data (precum Hadoop sau Spark), *MapReduce* este paradigma prin care se procesează volume uriașe de date prin împărțirea sarcinii în două faze:

- ***Faza Map:*** fiecare nod procesează o parte din date, aplicând operații de filtrare și transformare (similar cu metodele `filter`/`map` din exemplele noastre). Se generează astfel seturi intermediare de rezultate.
- ***Faza Reduce:*** rezultatele intermediare sunt apoi combinate și agregate de către noduri centralizatoare, într-un mod similar cu metoda `reduce` pe care am folosit-o pentru sumă. De exemplu, dacă am dori să calculăm suma unor valori dintr-un set de date masiv, fiecare nod ar calcula o sumă parțială (map), apoi aceste sume parțiale ar fi adunate împreună de reduce, obținând suma totală.

Acest model ***MapReduce permite procesarea în paralel*** a datelor masive și apoi reunirea rezultatelor, conceptul de bază fiind exact funcțiile *map* și *reduce* pe care le-am exersat la scară mică în JavaScript[7][8]. Așadar, ceea ce am învățat aici are aplicații directe în arhitectura algoritmilor de big data – de la indexarea paginilor web la agregarea de statistici din baze mari de date, toate se bazează pe împărțirea problemei în pași de tip map și reduce.

Referință [6]: Exemplele de la pasul 1 și 2 pun în evidență modul în care putem combina *filtrarea* cu *proiecția* și *agregarea* pentru a obține informația dorită din date brute[3][6]. Aceste tehnici stau la baza multor algoritmi eficienți de prelucrare a datelor.

PASUL 3 – Funcții variadice și formatare de șiruri

CONCEPT: O **funcție variadică** (engl. *variadic function*) este o funcție care poate accepta un **număr variabil de argumente** atunci când este apelată. Multe funcții uzuale sunt variadice. De exemplu, în JavaScript, `Math.max(x, y, z, ...)` poate primi oricâte numere și returnează pe cea mai mare – dacă îi dai 2 sau 10 argumente, le acceptă pe toate. Un alt exemplu clasic, din alte limbaje, este funcția `printf` din C, care ia un string de format și apoi un număr variabil de valori de inserat în acel șablon.

În JavaScript (ES6+), suportul pentru funcții variadice este oferit prin **parametrul *rest*** (`...numeParam`), care colectează toate argumentele suplimentare într-un array. De asemenea, înainte de ES6 exista obiectul special `arguments` disponibil în interiorul oricărei funcții (obișnuite) care conține toți parametrii transmiși (indiferent de numărul lor). În general, funcțiile variadice sunt utile când nu știm dinainte câte argumente vom primi sau vrem să oferim flexibilitate apelantului.

EXEMPLU PRACTIC (funcție de formatare): Să implementăm o funcție variadică simplă, `format(template, params)`, care primește un șablon de text (șir de caractere) ce conține **plasholders** (adică **locuri de inserare** marcate, de exemplu, cu forma `{nume}`) și un obiect de parametri cu valori pentru acei placeholderi, întorcând șirul formatat final. De exemplu, dacă apelăm `format("Hello {name}", {name: "Ana"})`, funcția ar trebui să returneze `"Hello Ana"`. În acest caz, funcția noastră primește un număr **variabil** de valori de înlocuit, în sensul că șablonul poate conține oricâte marcaje, iar obiectul de parametri poate avea oricâte proprietăți – funcția trebuie să se adapteze la ce i se dă.

Vom realiza formatarea prin **înlocuirea** fiecărui placeholder din text cu valoarea corespunzătoare din obiectul de parametri. O soluție simplă este să parcurgem cheile obiectului și să facem înlocuiri succesive în șablon.

```
function format(template, params) {
  let rezultat = template;
  for (const key in params) {
    // construim forma {cheie} care trebuie înlocuită
    const placeholder = "{" + key + "}";
    // înlocuim toate aparițiile (split & join pentru înlocuire globală)
    rezultat = rezultat.split(placeholder).join(params[key]);
  }
  return rezultat;
}

// Exemplu de utilizare:
console.log( format("Salut, {prenume} {nume}!", { prenume: "Ion", nume: "Popescu" }) );
// Output: "Salut, Ion Popescu!"
```

În codul de mai sus, funcția `format` ia fiecare cheie din obiectul `params` (de ex., "prenume", "nume") și înlocuiește toate aparițiile subșirului `{prenume}` din template cu valoarea corespunzătoare (ex. "Ion"), și la fel pentru `{nume}`. Am folosit o tehnică bazată pe `split/join` pentru a înlocui **toate** aparițiile unui

placeholder (în JavaScript, `String.replace` cu string normal înlocuiește doar prima apariție; am fi putut folosi și o expresie regulată cu indicator global, dar am preferat o metodă mai ușor de urmărit). La final, `rezultat` conține șirul formatat.

> ***Notă:*** *Funcția noastră nu este perfectă – de exemplu, dacă placeholderii se suprapun sau dacă o valoare de înlocuit este ea însăși ceva de forma `{...}`, pot apărea neajunsuri. În practică, formatarea robustă ar necesita un parser mai sofisticat. Totuși, pentru scopul didactic, acest exemplu arată clar conceptul: primim un șablon și un set arbitrar de perechi cheie-valoare, și le* ***interpolăm*** *într-un string.*

Pseudocod (înlocuirea placeholderilor într-un șablon):

```
rezultat = template
pentru fiecare pereche (cheie, valoare) din params:
    placeholder = "{" + cheie + "}"
    înlocuiește toate aparițiile lui placeholder în rezultat cu valoare
returnează rezultatul
```

Acest pseudocod reflectă direct implementarea. Observați că folosim un obiect `params` pentru flexibilitate – putem transmite oricâte perechi de înlocuire dorim, funcția le va procesa pe toate.

VARIADIC vs parametri ficsi/fixati: În exemplul nostru, funcția `format` are doi parametri formali (`template` și `params`), deci din punct de vedere strict nu pare "variadică" în definiție. Însă este **variadică logic**, întrucât al doilea parametru poate conține un număr variabil de valori (chei multiple în obiect). O altă abordare ar fi fost definirea funcției ca `function format(template, ...args)` și folosirea unui șablon de tip `{0}, {1}` pentru a insera `args[0]`, `args[1]` etc. (similar cum face `printf` cu `%s`, `%d`). De exemplu: `format("Hello {}, you have {} messages.", "Ana", 5)` ar putea itera peste `...args` și înlocui `{0}` cu "Ana", `{1}` cu 5 ș.a.m.d. Aceasta ar fi o implementare *posițională* variadică. Ideea principală este că putem trata argumentele suplimentare fie prin *nume* (ca în exemplul cu obiectul, mai ușor de citit) fie prin *poziție* (ca în exemplul cu `...args`).

ATENȚIE ȘI SFATURI:

- Folosirea obiectului pre-ES6 `arguments` într-o funcție variadică oferă acces la toate argumentele, dar amintește-ți că `arguments` este *pseudo-array* (nu are metode de array). O practică modernă este să folosești sintaxa `...` (rest parameters) care îți dă direct un array. De exemplu: `function variadicFunc(...args) { /* args este un [] */ }`.
- La formatarea stringurilor, ai grijă ca pentru volume mari de date (ex. generarea unui raport lung cu mii de înlocuiri) să folosești metode eficiente. În JavaScript, concatenarea simplă sau `replaceAll` sunt destul de eficiente pentru majoritatea cazurilor, dar dacă execuți milioane de înlocuiri (big data text processing), poate fi nevoie de algoritmi dedicați sau streaming (prelucrare text pe bucăți) pentru viteză și consum redus de memorie.
- Biblioteca standard JavaScript nu are o funcție de formatare a stringurilor cu placeholderi generic (în afară de template literals ES6 care folosesc sintaxa `${expr}` în stringuri delimitate de backtick ` – însă acelea evaluează expresiile direct, nu pe baza unui dicționar). Pentru formatare complexă, se folosesc adesea librării externe sau se implementează ad-hoc ca mai sus.

PASUL 4 – Implementarea funcțiilor `map` și `reduce` (extinderea prototipului Array)

Pentru a înțelege mai bine cum funcționează metodele `map` și `reduce` "sub capotă", vom realiza un exercițiu important: **implementarea manuală** (la scară mică) a acestor metode, și anume vom crea propriile versiuni `Array.prototype.myMap` și `Array.prototype.myReduce`. Aceasta ne va ajuta să

consolidăm înțelegerea conceptelor de callback, iterație și acumulare. În plus, vom vedea cum putem **extinde prototipul** obiectelor native JavaScript (deși în practică trebuie făcut cu grijă, e un procedeu util educativ).

Implementare `Array.prototype.myMap`:

Vom adăuga o funcție numită `myMap` în prototipul Array, astfel încât orice array să o poată folosi. `myMap` va primi un singur parametru: funcția de transformare (`callback`). Vom itera prin elementele array-ului (`this` se va referi la array-ul pe care e apelată metoda) și vom aplica `callback` pe fiecare element, colectând rezultatele într-un nou array pe care îl returnăm.

```
Array.prototype.myMap = function(callback) {
  const result = [];
  for (let i = 0; i < this.length; i++) {
    if (this.hasOwnProperty(i)) {           // opțional: verifică dacă indicele există
(pt. a sări peste eventuale elemente goale)
      result.push( callback(this[i], i, this) );
    }
  }
  return result;
};

// Exemplu de utilizare:
const arr = [1, 2, 3];
const dubluri = arr.myMap(x => x * 2);
console.log(dubluri);         // [2, 4, 6]
console.log(arr);             // [1, 2, 3] (array-ul original rămâne nemodificat)
```

Explicație: Metoda `myMap` de mai sus parcurge array-ul `this` de la indexul 0 la `this.length - 1` și aplică `callback(this[i], i, this)` pe fiecare element existent, stocând valoarea returnată de callback în noul array `result`. Am inclus și `if (this.hasOwnProperty(i))` ca măsură de siguranță pentru a ignora eventuale **goluri** în array (de ex., dacă ai un array cu lungime 5 dar numai 3 elemente setate, restul fiind „<empty>”, cum se întâmplă la anumite operații sparse – însă pentru array-uri normale nu e neapărat necesar). La final, `result` conține transformările și este returnat. Observați că `this` este folosit în interiorul unei funcții tradiționale `function(...) { }` – dacă am fi folosit sintaxa arrow (`=>`), `this` nu ar fi referit prototipul array (funcțiile săgeată nu au propriul `this`). De aceea, la definirea de metode de prototip preferăm sintaxa clasică pentru a avea acces corect la contextul obiectului.

Implementare `Array.prototype.myReduce`:

Acum implementăm `myReduce`, care trebuie să parcurgă array-ul și să **agregheze** valorile pe baza unui callback furnizat, asemenea `Array.prototype.reduce`. Aceasta e puțin mai delicată deoarece trebuie să ținem cont dacă s-a dat sau nu o valoare inițială și să iterăm corespunzător.

```
Array.prototype.myReduce = function(callback, initialValue) {
  let accumulator;
  let startIndex;
  if (initialValue !== undefined) {
    accumulator = initialValue;
    startIndex = 0;
  } else {
    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 (this.hasOwnProperty(i)) {        // asigură că elementul există (nu e hole)
```

```
      accumulator = callback(accumulator, this[i], i, this);
    }
  }
  return accumulator;
};

// Exemplu de utilizare:
const arr2 = [5, 7, 3];
const suma = arr2.myReduce((acc, val) => acc + val, 0);
console.log(suma);    // 15  (5+7+3, cu 0 ca inițial)
const produs = arr2.myReduce((acc, val) => acc * val);
console.log(produs);  // 105 (5*7*3, fără valoare inițială -> pornește de la 5)
```

Să analizăm pe scurt implementarea de mai sus:

- Dacă apelantul oferă o `initialValue`, atunci o folosim ca valoare de pornire a acumulării (`accumulator = initialValue`) și vom începe iterația de la primul element (index 0). Dacă **nu** se oferă valoare inițială, atunci trebuie să inițializăm acumulatorul cu **primul element al array-ului** (`this[0]`) și să începem iterarea de la al doilea element (index 1). În plus, dacă array-ul este gol și nu există valoare inițială, conform specificației standard trebuie aruncată o eroare (nu există nimic de redus) – am inclus acest caz printr-un `throw` corespunzător.
- Apoi iterăm prin array începând de la `startIndex` calculat. La fiecare pas, actualizăm `accumulator` apelând `callback(accumulator, elementCurent, indexCurent, this)`. Astfel, acumulatorul "adună" (sau combină) treptat valorile.
- La final, returnăm `accumulator`, care conține rezultatul agregării.

Exemplele de utilizare arată că metoda funcționează: pentru `suma`, am dat valoare inițială 0 și suma elementelor [5,7,3] a rezultat 15; pentru `produs`, nu am dat valoare inițială, deci `myReduce` a luat 5 (primul element) ca punct de start și apoi a înmulțit cu 7 și 3 succesiv.

> **De ce să implementăm manual astfel de metode?** Acest exercițiu ne ajută să înțelegem detalii subtile: cum se comportă `reduce` când nu primește valoare inițială, cum tratează elementele lipsite (găurile) din array, cum putem accesa contextul `this` al array-ului, etc. De asemenea, evidențiază conceptul de **funcții de ordin înalt**: atât `map` cât și `reduce` iau ca argument o altă funcție (`callback`-ul), ceea ce demonstrează că în JavaScript funcțiile sunt *cetățeni de primă clasă* (le putem pasa ca date, stoca în variabile, returna din alte funcții).

Atenționare: *În practică, extinderea obiectelor native (precum `Array.prototype`) poate fi riscantă, deoarece dacă două biblioteci diferite decid să adauge metode cu același nume, pot apărea ciocniri. În cazul nostru, am folosit nume distincte (`myMap`, `myReduce`) tocmai ca să evităm conflictul cu implementările standard existente. Este bine ca în codul de producție să fim precauți cu astfel de "monkey-patching". Scopul aici e didactic, pentru a vedea cum ar putea arăta implementarea internă a lui `map` și `reduce`. După cum se observă, logica internă nu este complicată, dar include tratamentul cazurilor speciale (gol vs inițializat, etc.), conform specificațiilor ECMAScript[1][2].*

Referință [7]: Acest pas evidențiază modul în care putem crea versiuni proprii ale unor metode de array, **pentru a înțelege cum funcționează la nivel intern** aceste metode predefinite[1][2]. Este o practică recomandată în învățare: încercați să reimplementați în cod propriu funcționalități ale limbajului, ca să le stăpâniți mai bine.

PASUL 5 – Cenzurarea cuvintelor și lanțuri de apeluri (method chaining)

Concept: *Cenzurarea* textului presupune înlocuirea anumitor cuvinte (de obicei considerate nepotrivite sau secrete) cu o formă mascată (de exemplu, cu asteriscuri "***"). Vom realiza o astfel de cenzurare folosind **lanțuri succesive de metode** pe string și array. Acesta este un exemplu

excelent de *method chaining*, unde rezultatul unei metode este folosit imediat pentru a apela o altă metodă, formând o succesiune fluentă de operații.

Planul este următorul: avem un text (șir de caractere) și o listă de cuvinte interzise. Vom **splitui** (divide) textul în cuvinte (pe baza spațiilor sau a semnelor de punctuație), obținând un array de "cuvinte". Apoi vom aplica `Array.map` pe acest array, înlocuind fiecare cuvânt care se află în lista nepermisă cu o variantă cenzurată (de ex. aceeași lungime de * sau un placeholder ca "[cenzurat]"). La final, vom **uni** (`join`) lista de cuvinte la loc într-un string unic. Toate aceste operații pot fi înlănțuite într-o singură expresie.

Exemplu practic (cenzurare text):

```
const text = "Este un text secret și acest secret trebuie cenzurat.";
const interzise = ["secret", "cenzurat"];  // cuvinte de cenzurat (case-insensitive)

// Aplica cenzura:
const textCenzurat = text
  .split(" ")  // împarte textul în cuvinte separate de spațiu
  .map(cuvant => {
      // eliminăm semnele de punctuație finale pentru verificare
      let cuvantCurat = cuvant.toLowerCase().replace(/[.,!?]$/g, "");
      if (interzise.includes(cuvantCurat)) {
          // înlocuiește cuvântul cu asteriscuri de aceeași lungime
          return "*".repeat(cuvantCurat.length) + (cuvant.endsWith('.') ? '.' : '');
      } else {
          return cuvant;
      }
  })
  .join(" ");
console.log(textCenzurat);
// Output: "Este un text ****** și acest ****** trebuie ******."
```

Să explicăm acest fragment pas cu pas:

- `text.split(" ")` separă șirul `text` în segmente, folosind spațiul ca delimitator. Rezultă un array de cuvinte (în exemplul nostru ar fi `["Este", "un", "text", "secret", "și", "acest", "secret", "trebuie", "cenzurat."]`).
- Pe acest array aplicăm `map`, procesând fiecare `cuvant`. În interiorul callback-ului:
    - Convertim cuvântul la lowercase și eliminăm eventuale semne de punctuație de la final (am folosit o expresie regulată simplă `/[.,!?]$/` care prinde ultimul caracter dacă e .,!,? sau ,). Facem asta pentru a putea verifica corect dacă cuvântul (fără punctuație și fără diferență de majuscule) se află în lista `interzise`. De exemplu, "secret." devine "secret" ca să fie recunoscut.
    - Verificăm dacă `cuvantCurat` (de ex. "secret") este inclus în array-ul `interzise`. Am definit `interzise` tot în lowercase, deci comparația e corectă indiferent de cum apare în text.
    - Dacă cuvântul este interzis, returnăm un șir de aceeași lungime cu `cuvantCurat` compus doar din `*`. Folosim `String.prototype.repeat` pentru a genera atâtea stelute câte litere are cuvântul original. Dacă cuvântul original se termina cu un punct (.), adăugăm punctul la final pentru a nu pierde punctuația în textul final (observați în cod: `+ (cuvant.endsWith('.') ? '.' : '')` – adaugă un punct dacă cuvântul original se termina în punct).
    - Dacă cuvântul nu e în lista interzisă, îl returnăm nemodificat.
    - După `map`, avem un array de cuvinte în care cele sensibile sunt înlocuite de asteriscuri. Apelăm `join(" ")` ca să reconstruim șirul complet, cu cuvintele separate prin spațiu. Rezultatul este stocat în `textCenzurat`.

În exemplul dat, `"secret"` (apărut de două ori) și `"cenzurat."` vor fi înlocuite cu `"******"`. Astfel, la afișare vedem că în locul acelor cuvinte apar grupuri de `*`, păstrând lungimea (6 litere în "secret" devin 6 asteriscuri, 7 în "cenzurat" devin 7 asteriscuri) și păstrând punctuația la final.

LANȚUL DE METODE (method chaining): Observați cum am scris codul pe mai multe linii, dar practic într-o singură expresie: `text.split(...).map(...).join(...)`. ***Acesta este un lanț de apeluri***. El este posibil deoarece fiecare metodă returnează un obiect pe care se poate invoca următoarea metodă: `split` a returnat un array (pe care putem apela `map`), iar `map` a returnat un array (pe care putem apela `join`). Avantajul chaining-ului este că putem gândi transformarea în pași și să le scriem direct ca atare, fără variabile intermediare (deși am fi putut folosi variabile pentru rezultate intermediare, nu era necesar). Un lanț de metode bine ales face codul mai **limpede**: citindu-l de la stânga la dreapta, vedem fluxul de procesare a datelor (întâi spargem textul, apoi transformăm fiecare cuvânt, apoi recompunem textul).

> ***Sfat:*** *Folosiți lanțuri de metode cu atenție. Dacă lanțul devine prea lung sau complicat, s-ar putea să fie mai greu de depanat sau de înțeles. În exemplul nostru, lanțul e scurt și logic. Totuși, dacă ai avea, să zicem, cinci sau șase transformări succesive, ar putea fi util să le spargi pe etape sau să comentezi fiecare etapă, pentru claritate. De asemenea, fiecare metodă din lanț alocă de obicei structuri noi (aici `split` alocă un array, `map` alocă alt array, `join` alocă un string). Pentru texte mici-medii acest lucru nu e o problemă, dar pentru texte uriașe (ex: fișiere de gigabyte) ar putea deveni ineficient ca memorie. În astfel de cazuri, o abordare streaming (procesarea bucăților de text secvențial) sau folosirea unor instrumente specializate de procesare text ar fi preferabilă.*

Referință [7]: Cenzurarea exemplificată aici demonstrează utilizarea lanțurilor de metode (`split -> map -> join`) pentru a realiza o prelucrare complexă într-o manieră succintă[9][10]. Prin astfel de transformări succesive, putem rezolva probleme ca aceasta într-un mod declarativ și ușor de urmărit.

PASUL 6 – Sortarea unui array de obiecte cu funcții de comparație

Concept: Sortarea elementelor este o operație fundamentală, iar metoda JavaScript dedicată este `Array.prototype.sort`. În forma sa simplă, `sort()` fără parametri, va ordona elementele array-ului **lexicografic** (ca și cum ar fi toate stringuri), ceea ce înseamnă că pentru numere, de exemplu, ordonarea va fi după valorile convertite la string (ceea ce poate produce rezultate ciudate, de exemplu [2, 10, 1] sortat lexicografic devine [1, 10, 2] fiindcă "10" < "2" lexical). Pentru a obține o ordonare numerică sau după un criteriu personalizat, `sort` acceptă ca argument o **funcție de comparație** (*comparison function* sau *comparator*). Această funcție decide ordinea a două elemente returnând un număr negativ, zero sau pozitiv, în funcție de relația dintre ele (negativ dacă primul trebuie să vină înaintea celui de-al doilea, pozitiv invers, iar zero dacă sunt considerate egale din punct de vedere al ordinii).

Vom vedea cum să sortăm un array de **obiecte** pe baza unui anumit **câmp (proprietate)** al obiectelor, folosind o funcție de comparație. De exemplu, vom avea o listă de persoane cu nume și vârstă, și vrem să le sortăm după vârstă crescător.

Exemplu practic (sortare după un atribut numeric):

```
const persoane = [
  { nume: "Ana", varsta: 30 },
  { nume: "Ion", varsta: 20 },
  { nume: "Maria", varsta: 25 }
];

// Sortare după vârstă (crescător):
persoane.sort((a, b) => a.varsta - b.varsta);
```

```
console.log("Ordonat după vârstă:", persoane);
/* Output:
Ordonat după vârstă: [
  { nume: "Ion", varsta: 20 },
  { nume: "Maria", varsta: 25 },
  { nume: "Ana", varsta: 30 }
] */
```

În funcția de comparație de mai sus, pentru două elemente a și b (care sunt obiecte de tip persoană), calculăm `a.varsta - b.varsta`. Dacă `a.varsta` este mai mică decât `b.varsta`, diferența va fi negativă, semnalând lui `sort` că a trebuie să vină înainte de b (ceea ce e corect, persoana mai tânără înaintea celei mai în vârstă). Dacă `a.varsta` > `b.varsta`, diferența va fi pozitivă și indică faptul că a ar trebui să vină după b. Dacă sunt egale (diferența 0), sort nu le va schimba între ele în raport unul cu altul (stabilitatea sortării standard în JS nu e garantată în specificație, dar majoritatea implementărilor mențin ordinea inițială pentru elemente egale).

Sortare descrescătoare: Pentru a inversa criteriul (de exemplu, dacă vrem persoanele de la cea mai vârstnică la cea mai tânără), putem inversa semnul sau ordinea scăderii:

```
persoane.sort((a, b) => b.varsta - a.varsta);  // comparație inversată
console.log("Ordonat descrescător după vârstă:", persoane);
```

Sortare după string (alfabetic): Dacă dorim să sortăm după nume (care sunt stringuri), compararea prin scădere nu se aplică direct. Trebuie fie să folosim metoda locale de comparare a stringurilor, fie să definim un comparator care întoarce -1, 0, 1. JavaScript oferă metoda `localeCompare` pentru stringuri, care returnează un număr negativ, zero sau pozitiv dacă stringul pe care e apelat e respectiv mai mic, egal sau mai mare (în ordine lexicografică) decât argumentul:

```
persoane.sort((a, b) => a.nume.localeCompare(b.nume));
console.log("Ordonat alfabetic după nume:", persoane);
/* Exemplu de output (dacă persoanele erau ca mai sus):
Ordonat alfabetic după nume: [
  { nume: "Ana", varsta: 30 },
  { nume: "Ion", varsta: 20 },
  { nume: "Maria", varsta: 25 }
] */
```

`localeCompare` are avantajul că știe să ordoneze corect și cu diacritice sau în alte limbi, conform regulilor locale, însă în ordinea generală "Ana" < "Ion" < "Maria" cum era de așteptat.

FUNCȚII DE COMPARAȚIE PERSONALIZATE: Putem scrie comparatoare mai complexe dacă dorim. De exemplu, sortare după mai multe criterii: întâi după varsta, apoi după nume dacă vârstele sunt egale. Asta s-ar implementa verificând în comparator: dacă `a.varsta - b.varsta` este 0 (adică aceeași vârstă), atunci returnăm `a.nume.localeCompare(b.nume)` ca să decide alfabetic; altfel returnăm diferența de vârstă.

IMPORTANT: `sort` **modifică array-ul original** (face sortarea in-place). Dacă aveți nevoie de versiunea sortată dar și de versiunea originală, clonați array-ul înainte de sortare (e.g. `const copie = persoane.slice(); copie.sort(...);`). Metoda returnează tot array-ul sortat, astfel că puteți eventual s-o folosiți și în lanțuri de apeluri, dar fiind o metodă cu efect in-place, de obicei nu se folosește în chaining decât dacă e ultimul pas sau efectul de modificare e intenționat.

ROL EDUCATIV: În acest pas am evidențiat modul de folosire a **funcțiilor ca parametri** (comparatorul transmis lui `sort`). Practic, i-am spus funcției sort *cum* anume să compare două elemente, definind o funcție la momentul apelului. Acesta e un exemplu clar de **funcție de nivel înalt**: `sort` este scris să apeleze funcția noastră de comparare oricând are nevoie să ordoneze două elemente. În plus, am recapitulat și că fără comparator, sortarea default e lexicografică, uneori surprinzătoare.

Performanță: Sortarea are o complexitate O(n log n) în medie. Pentru array-uri mari (sute de mii sau milioane de elemente), poate consuma timp notabil. În cazuri extreme (seturi de date foarte mari care nu încap în memorie), sortarea trebuie făcută fie în flux (parțial, cunoscut ca *external sorting*), fie delegată unei baze de date sau unui sistem distribuit. Dar pentru majoritatea aplicațiilor uzuale, `Array.sort` este optimizat și suficient de rapid în JavaScript.

PASUL 7 – Calcularea **mediei** elementelor unui array cu `reduce` (Opțional)

Ultimul nostru exemplu combină ceea ce am învățat despre `reduce` cu o operație aritmetică suplimentară pentru a calcula o valoare agregată mai complexă: **media aritmetică** a elementelor. Media unui set de valori se obține prin formula binecunoscută: sumă împărțită la număr de elemente. Putem calcula aceasta într-un mod foarte concis folosind `reduce`. Ideea de bază este să folosim reduce pentru a obține suma și apoi să împărțim la lungimea array-ului. Putem face asta fie **în două etape** (mai întâi reduce pentru sumă, apoi împărțire), fie chiar **într-o singură trecere** (de exemplu, detectând în callback când suntem la ultimul element și efectuând atunci împărțirea).

Exemplu practic (media numerelor dintr-un array):

```
const valori = [10, 20, 30, 40];
// Varianta 1: reduce + operație separată
const sumaTotală = valori.reduce((acc, x) => acc + x, 0);
const media = sumaTotală / valori.length;
console.log("Media (calcul în 2 pași):", media);  // 25

// Varianta 2: calculul mediei într-un singur reduce
const mediaIntrUnPas = valori.reduce((acc, x, index, arr) => {
  acc += x;
  if (index === arr.length - 1) {
    return acc / arr.length;
  } else {
    return acc;
  }
}, 0);
console.log("Media (un singur pas reduce):", mediaIntrUnPas);  // 25
```

Am prezentat **DOUĂ ABORDĂRI**: prima folosește două linii de cod (reduce pentru sumă, apoi împărțire), a doua folosește un *truc*: în interiorul lui `reduce` verificăm dacă am ajuns la ultimul element (`index === arr.length - 1`), caz în care returnăm direct `acc / arr.length` (calculul mediei), altfel continuăm să returnăm suma acumulată. La finalul rulării lui `reduce`, acumulantul va conține deja media (pentru că la ultima iterare am făcut împărțirea). Am inițializat acumulatorul cu 0 și am adunat fiecare x la el. Rezultatul este același, însă merită menționat că **varianta directă este mai puțin intuitivă** – de obicei e mai clar să separi conceptele: întâi suma, apoi media. Totuși, a doua variantă arată creativitatea cu care se pot combina operațiile în `reduce`.

Altă variantă: Putem folosi `reduce` și pentru a calcula **simultan** suma și numărul de elemente, stocând într-un acumulator de tip obiect ambele valori, apoi la final să calculăm media. De exemplu:

```
const rez = valori.reduce((acc, x) => {
  acc.sum += x;
  acc.count += 1;
  return acc;
}, { sum: 0, count: 0 });
const media2 = rez.sum / rez.count;
```

Aici am folosit acumulator un obiect cu două proprietăți, `sum` și `count`. La fiecare pas adunăm valoarea și incrementăm count. La final, calculăm media din aceste două. Aceasta

este o abordare clasică pentru situațiile în care vrei să agregi mai multe aspecte într-o singură trecere. E util, de pildă, dacă ai nevoie și de sumă și de count separat oricum.

Atenție: *Când calculezi media, trebuie să te asiguri că array-ul nu este gol (altfel împărțirea la 0 nu are sens). Dacă e posibil să fie gol, tratează separat cazul (de ex., returnează 0 sau* `null`*, sau aruncă eroare, după context). În exemplul nostru, știm că* `valori` *are elemente.*

Acest pas opțional a fost menit să arate cum putem aplica `reduce` și dincolo de suma simplă – **combinând** operația de reducere cu o operație aritmetică finală (împărțirea la număr de elemente). Practic, am evidențiat că `reduce` poate fi folosit nu doar pentru a acumula direct un rezultat final, ci și ca parte din calculul unei expresii mai complexe.

CE AM ÎNVĂȚAT (concluzii)

Parcurgând aceste exemple, am acumulat cunoștințe importante despre manipularea array-urilor și funcțiilor în JavaScript într-un stil modern, declarativ:

- Am văzut cum **`filter`** ne permite să extragem dintr-o colecție doar elementele care ne interesează, pe baza unei condiții logice definite de noi. Această metodă respectă principiul *filtrării* datelor și ne scutește de a scrie manual if-uri și de a gestiona indecși în mod explicit. Totodată, am înțeles că rezultatul filtrării este un nou array, originalul rămânând nemodificat[11][12].

- Am explorat metoda **`map`**, care aplică o funcție de transformare fiecărui element din array, obținând astfel o *proiecție* a datelor originale (de aceea lungimea rămâne neschimbată, dar conținutul se schimbă element cu element). Am folosit map pentru a atașa sufixe la numere, pentru a calcula pătrate, pentru a înlocui cuvinte cu asteriscuri etc. – demonstrând versatilitatea sa. Ca și filter, `map` creează un nou array, deci putem folosi rezultatele fără teama de a fi alterat array-ul inițial.

- Am combinat **`filter` și `map`** în lanțuri de apeluri, ceea ce ne-a permis să efectuăm **prelucrări secvențiale** pe date într-o singură expresie. Chaining-ul acestor metode face codul mai succint și mai ușor de urmărit, punând accent pe logica procesării datelor în locul detaliilor de implementare (bucle, indexare etc.).

- Am învățat să folosim **`reduce`** pentru a **agrega** un array la o singură valoare. Suma, produsul, maximul sau concatenarea stringurilor sunt doar câteva exemple; am mers și mai departe, calculând media, ceea ce a implicat combinarea lui reduce cu operații adiționale (împărțire). Am subliniat importanța alegerii corecte a valorii inițiale și modul în care reduce se comportă când aceasta lipsește.

- Ne-am aplecat asupra **funcțiilor variadice**, definind o funcție care acceptă parametri flexibili pentru a formata un șir de caractere. Am discutat atât vechiul mecanism cu `arguments`, cât și noul `...rest` din ES6, și am implementat o soluție simplă de **interpolare** a unui șablon text cu valori, întărind astfel ideea de funcție reutilizabilă și flexibilă.

- Prin implementarea propriilor versiuni **`myMap` și `myReduce`**, am înțeles mai bine detaliile interne ale acestor metode de array. Am recunoscut că ele nu sunt magie, ci parcurg elementele și apelează callback-ul nostru, construind un rezultat. Acest lucru a consolidat conceptele de *callback*, *this*, acumulare, gestionarea valorii inițiale etc. De asemenea, am experimentat extinderea prototipurilor și am discutat despre implicațiile acestui procedeu.

- Am exersat folosirea **lanțurilor de metode** nu doar pentru filtrare și mapare, ci și pe stringuri, combinând `split`, `map` și `join` pentru a rezolva o problemă practică (cenzurarea cuvintelor

interzise). Astfel de lanțuri evidențiază puterea combinatorie a metodelor: fiecare tranformare se aplică peste rezultatul celei anterioare, permițând soluții elegante pentru probleme altfel rezolvabile prin cod mai stufos.

- Am utilizat **`sort` cu comparator** pentru a ordona obiecte, evidențiind încă o dată modul în care putem personaliza comportamentul unei funcții de bibliotecă trecând o altă funcție ca parametru. Am discutat despre sortarea numerelor vs stringuri, criterii multiple și faptul că sortarea are loc in-place.

În esență, am pus accentul pe un stil de programare care favorizează **expresivitatea**: a spune "ce vrem să obținem" mai degrabă decât "cum să iterăm pas cu pas". Acest stil declarativ, centrat pe funcții ca `filter`, `map`, `reduce`, duce de obicei la cod mai **scurt**, mai **clar** și mai ușor de modificat. Desigur, este important să știm și ce se întâmplă în culise (complexitatea, impactul în memorie, limitările), dar odată stăpânite, aceste unelte devin extrem de utile pentru orice dezvoltator.

ATENȚIONĂRI ȘI SFATURI

- **EVITAȚI EFECTELE SECUNDARE NEDORITE:** Metodele `filter`, `map`, `reduce` nu modifică array-ul sursă (cu excepția sortării, care modifică in-place). Profitați de acest lucru pentru a scrie cod fără efecte secundare. Dacă totuși modificați elementele în interiorul callback-urilor (de exemplu, dacă un element e obiect și îi schimbați o proprietate în `map`), fiți conștienți că afectați structurile existente. Ideal, funcțiile de callback ar trebui să fie *pure* (fără efecte secundare) pentru a menține codul previzibil.

- **REDUCE ȘI VALORILE INIȚIALE:** Întotdeauna întrebați-vă ce se întâmplă dacă array-ul de intrare ar fi gol. Dacă e posibil să fie gol, furnizați o valoare inițială la `reduce` pentru a evita erori. De exemplu, `[].reduce(...)` fără inițială va arunca excepție. Dacă calculul nu are sens pe date goale, poate e mai bine să verificați separat și să tratați cazul (ex: returnați 0 sau `null` pentru media unei liste goale, după cum e logic în context).

- **PERFORMANȚĂ vs LIZIBILITATE:** Metodele funcționale și lanțurile de apel fac codul lizibil, dar pot crea array-uri intermediare suplimentare (cum am menționat la cenzurare, `split->map->join` implică două array-uri intermediare și un final string). De obicei acest overhead este neglijabil față de avantajul clarității. Totuși, în cazuri critice (ex: bucle în milioane de iterări), merită făcute teste de performanță. Uneori o combinație de operații poate fi înlocuită cu o singură trecere (de ex., în loc de `filter` + `map` separate, se poate face totul într-un singur `reduce` dacă e nevoie absolută de eficiență, sacrificând din claritate). Optimizează abia după ce identifici un motiv real (principiul "premature optimization is the root of all evil").

- **EXTENSIA PROTOTIPURILOR:** După cum am discutat, extinderea prototipurilor obiectelor native (Array, Object, etc.) trebuie făcută cu grijă. În medii controlate (de exemplu, la exerciții sau dacă implementezi *polyfill*-uri când o funcționalitate nu este disponibilă nativ), e în regulă. Dar în aplicații mari, există riscul ca două coduri diferite să creeze metode cu același nume sau ca viitoare standarde să introducă o metodă cu numele pe care tu l-ai ales, provocând confuzie. Un sfat este să folosești prefixe unice dacă vrei neapărat să extinzi (ex: `myMap` în loc de `map` deja existent).

- **COMPARATORUL LUI `sort`:** Asigură-te că funcția de comparație este **consistentă**: dacă a trebuie egal cu b, returnează 0; dacă a < b returnează mereu o valoare negativă, și invers. Un comparator scris incorect (care nu respectă tranzitivitatea, de pildă) poate duce la rezultate imprevizibile, deoarece algoritmul de sortare depinde de aceste contracte. De asemenea, nu te baza pe sortarea fiind stabilă (deși V8 în Chrome, de exemplu, are sortare stabilă din ES2019 pentru elemente cu comparator ce returnează 0, acest lucru nu era garantat în standard înainte).

- **BIG DATA ȘI PROCESAREA FUNCȚIONALĂ:** Dacă vreodată vei lucra cu date foarte mari, amintește-ți că tehnicile de mai sus se extind conceptual. Biblioteca sau limbajul folosit ar putea avea funcții echivalente ce operează *lazy* (fără să genereze imediat liste intermediare) sau paralel. De exemplu, în Python există generatori și expresii generatoare care fac ceva similar cu map/filter dar pe rând, fără a ține totul în memorie, sau în Java 8+ există *Streams* care permit operații funcționale și se pot executa și concurent. Iar la nivel de infrastructură, cum am menționat, există Hadoop MapReduce, Apache Spark și altele care implementează la scară largă aceste paradigme. Cu alte cuvinte, gândirea în termeni de map-filter-reduce îți va fi utilă și dincolo de JavaScript, și pentru seturi de date de orice dimensiune.

În concluzie, adoptarea stilului funcțional în manipularea array-urilor poate aduce un plus de claritate și concizie codului vostru. Ca dezvoltatori, e bine să fim familiarizați cu aceste instrumente și să știm când să le aplicăm pentru a scrie cod elegant, dar și să fim conștienți de mecanismele lor interne și implicațiile asupra performanței.

QUIZ (cu răspunsuri explicate)

Pentru a recapitula cunoștințele dobândite, iată un scurt quiz. Fiecare întrebare este urmată de răspunsul explicat:

Întrebarea 1: Ce fac metodele `filter` și `map` asupra unui array? Modifică ele array-ul original sau creează unul nou? Explicați pe scurt.

Răspuns: `filter` parcurge un array și **selecționează** elementele care îndeplinesc o anumită condiție (funcția de filtrare returnează true pentru ele), întorcându-le într-un **array nou**. `map` parcurge un array și **transformă** fiecare element prin funcția dată, rezultând de asemenea un **array nou**, de aceeași lungime cu cel original (fiecare element este înlocuit de rezultatul funcției de mapare). **Nici `filter`, nici `map` nu modifică array-ul original** – ele creează array-uri separate cu rezultatele respective[13][14]. Originalul rămâne intact, ceea ce ajută la evitarea efectelor secundare. (Notă: `sort` în schimb modifică originalul, dar `filter`/`map` nu).

Întrebarea 2: Cum se comportă `reduce` dacă apelăm metoda pe un array gol fără să furnizăm o valoare inițială? Dar dacă furnizăm o valoare inițială?

Răspuns: Dacă apelăm `[].reduce(callback)` **fără valoare inițială**, JavaScript va arunca o eroare de tip **TypeError** (deoarece nu există elemente din care să scoată o valoare inițială implicită). Practic, `reduce` nu are niciun element de pornire. În schimb, dacă furnizăm o **valoare inițială**, de exemplu `[].reduce(callback, valoareInit)`, atunci pentru un array gol `reduce` va returna pur și simplu `valoareInit` fără să apeleze deloc callback-ul (pentru că nu are ce itera). În cazul unui array ne-gol,

dacă dăm valoare inițială, reduce începe acumularea de la acea valoare; dacă nu dăm, reduce va lua primul element al array-ului ca acumulator inițial și va începe de la al doilea element iterarea. Așadar, **valoarea inițială este opțională dar importantă**: previne erorile pe array-uri vide și poate influența tipul rezultatului (de exemplu, `[].reduce((a,b)=>a+b, 0)` dă 0 pe array gol, în loc să dea eroare)[15][16].

Întrebarea 3: Explicați ce este o funcție variadică și dați un exemplu (din ce am discutat sau din biblioteca JavaScript).

Răspuns: O funcție variadică este o funcție ce poate primi un **număr oricât de mare de argumente**, diferit de la un apel la altul. Nu are un număr fix de parametri. Un exemplu din JavaScript este `Math.max()`: poți apela `Math.max(1,5,10)` sau `Math.max(7, -2, 4, 11, 0)` etc., iar funcția va procesa toate argumentele date (în acest caz returnează cea mai mare valoare din listă). Alt exemplu ar fi funcția noastră `format(template, params)` discutată la Pasul 3, care prin obiectul `params` poate acomoda oricâte înlocuiri – deci este variadică logic. Dacă ne gândim la sintaxa modernă, orice funcție definită cu `...rest` devine variadică. Exemplu: `function f(...args) { console.log(args.length) }` – o putem apela cu oricâți parametri, iar în interior `args` va fi un array ce le conține. Conceptual, funcțiile variadice sunt utile pentru operații ca sumarea unei liste de numere (`sum(1,2,3,4)`) sau construirea unui șir formatat (`printf("%d %s", 5, "elemente")` ș.a.). În rezumat, o funcție variadică **se adaptează la numărul de argumente** cu care este apelată.

Întrebarea 4: Cum se poate sorta un array de obiecte după o anumită proprietate (de tip numeric) folosind JavaScript? Care este rolul funcției de comparație în acest proces?

Răspuns: Pentru a sorta un array de obiecte după o proprietate numerică, folosim metoda **`sort` cu o funcție de comparație personalizată**. De exemplu, dacă avem `obj.prop` numeric, apelăm `array.sort((a, b) => a.prop - b.prop)`. Funcția de comparație primește două elemente (`a` și `b`) și trebuie să returneze: un **număr negativ** dacă dorim ca `a` să apară înainte de `b`, un **număr pozitiv** dacă `a` trebuie să apară după `b`, sau **0** dacă sunt echivalente ca ordine. În exemplul `a.prop - b.prop`, practic scăderea va da un rezultat negativ atunci când `a.prop` < `b.prop` (deci `a` vine înainte, cum vrem pentru ordonare crescătoare), pozitiv când `a.prop` > `b.prop` (deci `a` vine după) și zero când sunt egale (lăsând eventual ordinea originală între ele). Această funcție de comparație ghidează algoritmul de sortare în efectuarea comparațiilor între elemente. Fără comparator, `sort` ar converti elementele la string și le-ar ordona lexicografic, ceea ce nu ar fi corect pentru numere sau pentru ordonare pe o proprietate specifică. Deci rolul comparatorului este esențial: **definește criteriul de ordonare** după care `sort` va aranja elementele.

Întrebarea 5: Ce este paradigma MapReduce în contextul Big Data și cum se leagă de metodele map și reduce studiate?

Răspuns: *MapReduce* este un model de programare pentru procesarea unui volum foarte mare de date în mod distribuit (pe mai multe mașini/noduri de calcul). Numele provine chiar de la cele două etape principale ale procesului: **Map** și **Reduce**. În faza de **Map**, datele sunt împărțite pe fragmente și fiecare fragment este prelucrat independent – se aplică operații de filtrare, transformare, mapare (de exemplu, extragerea unor perechi cheie-valoare relevante din datele brute). Apoi vine faza de **Reduce**, unde rezultatele intermediare produse de map sunt combinate/aggregate pentru a obține rezultatul final (de exemplu, sumarizarea contorizărilor, gruparea pe chei, etc.). Legătura cu metodele studiate este directă: conceptul de "map" din MapReduce corespunde ideii de a aplica o funcție (de transformare) fiecărei unități de date (similar cum metoda `map` aplică o funcție fiecărui element al

array-ului). Iar conceptul de "reduce" corespunde ideii de a aduna toate aceste rezultate într-unul singur (exact cum metoda `reduce` ne permite să agregăm un array la o valoare). Diferența e că în Big Data, aceste operații sunt realizate în paralel, la nivel de cluster, nu doar în memoria unui singur program, dar principiile sunt aceleași. Așadar, ceea ce am practicat cu `map` și `reduce` în JavaScript la nivel de array se generalizează în MapReduce ca model pentru a procesa eficient cantități masive de date împărțite pe mai multe calculatoare[7][8]. Este o dovadă a puterii acestor abstracțiuni că se aplică de la colecții mici în memorie până la dataset-uri de petabytes pe clustere distribuite.

BIBLIOGRAFIE:

Referințe: [6] și [7] corespund materialelor didactice din platforma neXtlab ASE, care acoperă în detaliu principiile de filtrare, proiecție și reducere, precum și implementarea și utilizarea avansată a acestor metode în JavaScript. Aceste resurse au stat la baza exercițiilor discutate și pot fi consultate pentru o aprofundare teoretică și practică suplimentară[3][17]. De asemenea, documentația oficială MDN pentru metodele Array și lucrările despre MapReduce oferă informații extinse despre subiectele atinse.

THE END IS NOT HERE: Înarmat cu aceste cunoștințe, vă încurajăm să exersați pe cont propriu: luați diferite probleme (cum ar fi transformarea listelor, agregarea datelor din obiecte, prelucrări de text) și încercați să le rezolvați folosind `filter`, `map`, `reduce` și celelalte tehnici prezentate. Prin practică, veți dobândi fluență în gândirea funcțională și veți putea scrie cod din ce în ce mai elegant și eficient. Succes!

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [15] [16] [17] din [https://student.nextlab.tech -> Tehnologii Web -> Funcții și array-uri] dar si din *S2verC (nextlab) Funcții și Array-uri în JavaScript – Prelucrare Funcțională a Datelor.docx*

[11] [12] [13] [14] din *S2verB (nextlab but simplier) Funcții și Array-uri în JavaScript.docx*