FUNCȚII ȘI ARRAY-URI ÎN JAVASCRIPT

1. INTRODUCERE

Acest material consolidează cunoștințele despre **funcțiile** și **tablourile (array-urile)** din JavaScript, punând accent pe practicile moderne ES6+. Vom examina funcțiile cu număr variabil de parametri (variadice) și modul în care putem prelucra colecții (șiruri de caractere, array-uri, obiecte) folosind abordări imperative și funcționale. Exemplele practice includ filtrarea cuvintelor, calculul proprietăților unui pătrat, formatarea output-ului, înmulțirea elementelor unui array, generarea unui acrostih și filtrarea proprietăților unui obiect. Fiecare exemplu este prezentat în mai multe variante (de la soluții clasice la unele idiomatice modern JS), evidențiind diferențe de paradigmă, lizibilitate și eficiență. Materialul continuă firul didactic al introducerii în JavaScript și integrează explicații teoretice cu scripturi demonstrative.

2. OBIECTIVE

La finalul acestei secțiuni, veți fi capabili să:

- Definiți și utilizați **funcții variadice** în JavaScript, comparând abordarea tradițională (obiectul `arguments`) cu sintaxa modernă (parametrul *rest* `...args` și operatorul *spread*).
- Înțelegeți diferența dintre obiectul special `arguments` și parametrii *rest*, precum și limitările funcțiilor *arrow* (lipsa propriului `this` și a propriului `arguments`[1]).
- Aplicați atât **paradigma imperativă** (bucle clasice, mutația in-place a array-urilor) cât și **paradigma funcțională** (metode array precum `filter`, `map`, `reduce` care returnează array-uri noi) pentru a rezolva probleme concrete de procesare a datelor.
- Evaluați compromisurile între lizibilitate și performanță: de exemplu, folosirea metodelor care creează array-uri noi vs. modificarea in-place a celor existente (imutabilitate vs eficiența memoriei).
- Utilizați structuri de control moderne în lucrul cu colecții: iterația cu `for...of` (pentru elemente ale array-ului) vs `for...in` (pentru cheile unui obiect), metode de iterare precum `forEach`, și funcții de ordin înalt (`filter`, `map`, etc.).
- Integrați scripturile JavaScript cu mediul de execuție Node.js (CLI) acolo unde este necesar, citind argumente din linia de comandă și afișând rezultate formatate.

3. PRECONDIȚII și mediu de lucru

Pentru a rula exemplele, este necesar un mediu de dezvoltare configurat corespunzător. Asigurați-vă că aveți **Node.js** instalat (versiunea LTS curentă este recomandată) și un terminal (Command Prompt/PowerShell pe Windows, Terminal pe macOS/Linux). Un editor de cod precum **Visual Studio Code** este util pentru scrierea și salvarea scripturilor înainte de rulare. Familiaritatea cu rularea unui fișier `.js` în Node (ex.: comanda `node scriptul_meu.js`) este presupusă. Cunoștințe prealabile necesare includ: sintaxa de bază JS (declarații, operatori), utilizarea `console.log` pentru output, și accesarea argumentelor din linia de comandă prin `process.argv`. În exemple, vom menționa uneori și facilități disponibile doar în versiuni moderne de Node/JS (de ex. Unicode property escapes în regex necesită Node 12+).

4. FUNDAMENTE TEORETICE – Funcții variadice și prelucrarea colecțiilor

În această secțiune revizuim conceptele teoretice ce stau la baza exemplelor practice. Vom începe cu funcțiile variadice și noțiuni asociate, apoi vom discuta abordările imperativă vs. funcțională în manipularea array-urilor, operatorii utili (`...` spread/rest) și iterația modernă.

4.1 Funcții variadice:

O funcție **variadică** este acea funcție care poate primi un număr arbitrar de argumente (parametri efectivi) la apel. Aceste funcții sunt utile pentru operații de agregare unde nu se cunoaște dinainte câte valori vor fi combinate (de exemplu, sumarea unui set de numere, concatenarea mai multor șiruri sau combinarea elementelor din array-uri). În JavaScript, spre deosebire de alte limbaje, toate funcțiile sunt în esență variadice la nivel de apel – adică se poate apela o funcție cu mai mulți parametri decât a declarat formal. Ce se întâmplă cu parametrii extra? În funcțiile obișnuite (non-**arrow**), ei pot fi accesați printr-un obiect special numit `arguments`. Vom detalia acest aspect în secțiunile următoare.

4.2 Obiectul `arguments` (pre-ES6):

În funcțiile declarate clasic (`function f(...) { ... }`), JavaScript pune la dispoziție local un obiect asemenător unui array, numit **`arguments`**, care conține toate valorile argumentelor primite la apel, indiferent de numele parametrilor formali. `arguments[0]` corespunde primului argument, `arguments[1]` celui de-al doilea ș.a.m.d., iar proprietatea `arguments.length` indică numărul total de argumente. Acest obiect este *array-like*, având indici și `length`, dar **nu este un Array veritabil**; de exemplu, nu are metode precum `forEach` sau `filter` decât dacă îl convertim explicit (ex.: `Array.from(arguments)` sau folosind operatorul spread: `[...arguments]`). De asemenea, în modul strict, `arguments` nu reflectă în mod sincronizat modificarea parametrilor nominali (în non-strict mode, există un comportament istoric de *aliasing* între parametri și entries din `arguments`, care însă dispare în strict mode[2]). Obiectul `arguments` este implicit în fiecare funcție obișnuită, dar **nu este disponibil în funcțiile *arrow*** (acestea nu au propriul `arguments`[1]). Pentru funcțiile *arrow*, dacă avem nevoie să accesăm argumentele, fie folosim variabile din context (o funcție *arrow* preia `arguments` de la funcția înconjurătoare dacă există), fie – mult mai recomandat – utilizăm parametrii *rest*, discutați mai jos.

4.3 Parametrul *rest* `...args` (ES6+):

Sintaxa de **parametru *rest*** introduce o modalitate clară și comodă de a indica faptul că o funcție poate accepta un număr variabil de argumente. În definiția funcției, ultimul parametru poate fi prefixat cu `...`, de exemplu: `function f(a, ...args) { }`. Acest parametru va fi automat un *Array* ce conține toate argumentele suplimentare furnizate la apel, dincolo de cele corespunzătoare parametrilor nominali precedenți. Astfel, *rest* oferă un mod simplu de a scrie funcții variadice moderne. Spre deosebire de `arguments`, parametrii *rest* sunt deja array-uri reale, deci putem aplica direct metode ca `sort`, `map`, `pop` etc.[2].

Există și alte diferențe importante față de `arguments`:

(1) `arguments` include **toate** argumentele, pe când parametrul *rest* colectează doar pe cele **rămase** (ex. în `function f(x, ...rest)`, `x` va lua primul argument, iar restul merg în array-ul `rest`);

(2) `arguments.callee` (referința către funcția curentă) există pentru `arguments` (deși este *deprecated*), dar nu are echivalent cu *rest*;

(3) în funcții non-strict clasice, `arguments` reflecta modificarea parametrilor, pe când array-ul *rest* nu se va sincroniza cu parametri (comportament oricum irelevant în strict mode);

(4) sintaxa *rest* oferă direct un **Array** și este mai sigură și mai expresivă. În concluzie, parametrul *rest* este modul recomandat de a scrie funcții variadice moderne în JS[3][2].

Exemplu: O funcție variadică de insumare poate fi definită elegant astfel:

```
function sum(...nums) {
  return nums.reduce((total, x) => total + x, 0);
}
console.log(sum(1, 2, 3, 4)); // -> 10
```

Aici `...nums` colectează toate argumentele într-un array `nums`, peste care apoi aplicăm `reduce` pentru a calcula suma. Dacă foloseam `arguments`, implementarea ar fi fost mai verbuoasă (ar fi necesitat conversie la array sau iterare manuală).

4.4 Operatorul *spread* `...`:

Operatorul **spread** utilizează aceeași notație `...` dar în context opus parametrului rest. În loc să colecteze valori, *spread* **expandează** o colecție iterabilă (de ex. un array sau string) în elementele sale componente.

Poate fi folosit:

(1) la apelul unei funcții, pentru a pasa elementele unui array drept argumente separate (echivalent modern pentru `func.apply`) – ex: `Math.max(...[4, 7, 1])` este echivalent cu `Math.max(4, 7, 1)`;

(2) la construirea unui array literar – ex: `let c = [...a, ...b]` concatenează array-urile a și b;

(3) la construirea unui obiect literar – ex: `let obj3 = { ...obj1, ...obj2 }` combină proprietățile (atenție, nu este pentru iterabile aici, ci copiază propriile proprietăți enumerabile ale obiectelor).

În esență, *spread* este **complementar** cu *rest*: *spread* "desface" o colecție în elementele sale, pe când *rest* "adună" elemente disparate într-o colecție[4]. Această paralelă ajută la reținerea modului de utilizare corect: de exemplu, într-o definiție de funcție `function f(x, ...rest)`, `...rest` colectează argumentele excedente într-un array; invers, la apel `f( ...arrayDeArgumente )`, operatorul *spread* împrăștie (expandează) elementele acelui array ca argumente individuale.

4.5 Funcții *arrow* (=>):**

Funcțiile săgeată introduse în ES6 oferă o sintaxă mai concisă și un comportament lexical pentru `this`. O **funcție arrow** are forma generală: `(param1, param2, ...) => expresie` (sau bloc de cod cu `{}`), fiind utilă mai ales pentru funcții mici sau ca argument pentru metode de array (`map`, `filter` etc.). Particularitatea majoră a funcțiilor *arrow* este că *nu* au propriul context `this` – ele preiau (`this` și `arguments`) din *scopul lexical* (contextul în care sunt definite)[1]. Astfel, într-o funcție arrow nu putem utiliza `arguments` (dacă încercăm, se va referi la `arguments` din funcția părinte, dacă există). De asemenea, funcțiile arrow nu pot fi folosite ca *constructori* (nu au proprietatea `prototype`). În schimb, ele permit o sintaxă foarte succintă: dacă corpul este o singură expresie, rezultatul acesteia este returnat implicit, fără a scrie `return`. Exemplu comparativ:

```
// Funcție clasică
function mul(a, b) {
  return a * b;
}
// Funcție arrow echivalentă
const mul2 = (a, b) => a * b;
```

Ambele pot fi apelate cu aceleași argumente. În contexte de callback, funcțiile arrow elimină necesitatea de a scrie funcții anonime verbose. Vom vedea numeroase exemple în care folosim sintaxa `=>` pentru simplitate. Este important de subliniat că, dacă avem nevoie într-adevăr de obiectul `arguments` sau de propriul `this` al funcției, **nu** vom folosi o arrow function, ci o funcție clasică. În caz contrar (majoritatea situațiilor de callback), arrow este preferată pentru claritate.

4.6 Imperativ vs. funcțional – Mutație in-place vs. imutabilitate:

În lucrul cu array-uri, există adesea două abordări posibile:

- **Stilul imperativ, cu mutație in-place:** se modifică direct structura de date existentă, de obicei folosind bucle sau metode care mută array-ul original. Avantajul este eficiența spațială (nu se alocă un nou array, se lucrează O(1) memorie în plus) și adesea claritatea algoritmică pas-cu-pas. Dezavantajul poate fi că se modifică datele de intrare, ceea ce în contexte mai complexe (sau reactive) poate duce la efecte secundare nedorite.

- **Stilul funcțional, cu structuri noi (immutability):** în loc să modificăm array-ul inițial, se creează un **nou array** cu rezultatele dorite (prin metode ca `filter`, `map`, concatenare cu `...`, etc.). Acest stil evită efectele secundare (originalul rămâne nemodificat) și poate fi mai ușor de testat și de paralelizat, dar costă în general timp și memorie adițională (complexitate O(n) pentru a crea noul array cu *n* elemente, plus eventual overhead de garbage collection pentru cel vechi). De exemplu, comparând două variante de a adăuga elemente la un array: una *imperativă* ar fi folosind `array.push(elem)` care modifică `array` existent (cost amortizat O(1) pe element adăugat) vs. una *funcțională* folosind operatorul spread: `[...array, elem]`, care creează un nou array (copiază toate elementele existente și adaugă la final – cost O(n)). Ambele au utilizări valide: prima e preferată când eficiența e critică și efectul de mutație este local și controlat; a doua e preferată când vrem transparență și evitarea modificării stării existente (ex. în programe React, Redux, etc., unde se favorizează actualizări imuabile).

Ca idee, programatorul JS trebuie să cunoască ambele stiluri! Vom evidenția în exemplele următoare ce variantă mută obiectul original și care construiește obiecte noi, precum și implicațiile de complexitate în timp și spațiu pentru fiecare soluție.

> ***Notă:*** *JavaScript oferă numeroase metode de array iterative (care parcurg elementele intern): `forEach`, `every`, `some`, `filter`, `map`, `reduce` etc. Acestea asigură un stil declarativ, unde specificăm ce dorim să facem cu elementele, nu cum să iterăm exact. Ele nu modifică de regulă array-ul original (cu excepția unor metode mutative precum `sort`, `reverse`, `splice` ș.a.), ci produc rezultate noi. În schimb, folosirea explicită a buclelor (`for`, `while`) este mai apropiată de metal, necesitând gestionarea indicilor, a condițiilor de oprire etc., dar permițând optimizări specifice (oprirea timpurie din buclă, mutații in-place ș.a.). Alegerea depinde de context: într-un cod ce favorizează claritatea și imutabilitatea, metodele funcționale sunt ideale; într-un algoritm critic de performanță sau foarte procedural, buclele imperative pot fi mai potrivite.*

4.7 Iterarea modernă a colecțiilor:

Iterarea peste elementele unui container a devenit mai simplă odată cu introducerea instrucțiunii **`for...of`** (ES6). În cazul unui array `arr`, expresia `for (const element of arr) { ... }` parcurge direct *valorile* din array, în ordine, asignându-le rând pe rând variabilei `element`. Aceasta evită scrierea unor indexări precum `for(let i=0; i< arr.length; i++) { let element = arr[i]; ... }`. De asemenea, `for...of` este **sigură pentru iterarea string-urilor Unicode**, întrucât parcurge caracterele complete (punctele de cod Unicode), nu unitățile de cod UTF-16 individuale (cum face accesarea cu index sau `for...in` pe string)[5][6]. În contrast, **`for...in`** se folosește în principal pentru obiecte – el iterează *cheile (numele proprietăților)* unui obiect. Dacă îl aplicăm unui array, `for...in` va parcurge indicii (ca string-uri) ai elementelor definite în array – ceea ce nu este de obicei

ceea ce dorim pentru procesarea valorilor. Regula practică: pentru array-uri (și alte iterabile precum Set, Map, argumente etc.) se folosește `for...of`, iar pentru obiecte simple (dicționare) `for...in`. Vom vedea exemple de ambele în secțiunea practică. Tot pentru obiecte, merită amintite metodele statice `Object.keys(obj)` – care returnează un array cu toate cheile proprii ale obiectului, `Object.values(obj)` – array cu valorile corespunzătoare, și `Object.entries(obj)` – array de perechi `[cheie, valoare]`. Acestea permit combinarea cu metodele de array (`forEach`, `filter`, `map` etc.) pentru a itera și prelucra eficient conținutul obiectelor.

Dupa ce ne-am reamintit cele de mai sus, să trecem acum la **exemplele practice**, care vor ilustra prin cod și explicații conceptele discutate. Fiecare exemplu propune o **sarcină** și multiple **variante de rezolvare** (notate A, B, C, ...), însoțite de discuții. Pagini noi au fost inserate înaintea fiecărei secțiuni de exerciții pentru claritate în formatul tipărit.

5. Filtrarea cuvintelor (WordsFilter A–F)

Cerința: Să se scrie o funcție care primește un șir de text (fraza) sau o listă de cuvinte și **returnează toate cuvintele care au lungimea de cel puțin 3 caractere** (presupunem că "cuvintele" sunt separate de spațiu în text). Cu alte cuvinte, se filtrează din text cuvintele "scurte" (sub 3 litere). Vom rezolva această problemă în 6 variante, evidențiind diferite stiluri de iterare și de folosire a facilităților limbajului.

5.1. **Var. A** — *wordsfilterA.js* (imperativ, buclă `for` clasică)

```
function filterWordsLongerThan3(text) {
  const words = text.split(' ');
  const result = [];
  for (let i = 0; i < words.length; i++) {
    if (words[i].length >= 3) {
      result.push(words[i]);
    }
  }
  return result;
}

// Exemplu de utilizare:
console.log(filterWordsLongerThan3("ana are mere și prune"));
// -> [ 'mere', 'prune' ]
```

Explicație: Funcția `filterWordsLongerThan3` împarte textul primit în array-ul de cuvinte `words` (folosind metoda simplă `String.split(' ')` pe spațiu) și parcurge acest array cu o buclă indexată clasic. Pentru fiecare cuvânt, verifică lungimea (`words[i].length`) și, dacă este de cel puțin 3, îl adaugă la array-ul `result`. La final, `result` (care conține doar cuvintele suficient de lungi) este returnat. Această variantă este **imperativă** și **mutativă** în sensul că umple treptat un array rezultat. Complexitatea timp este O(n) (unde n este numărul de cuvinte), iar complexitatea spațială O(n) pentru array-ul rezultat (măsurând spațiul adițional față de intrare).

Comentarii: Metoda `split(' ')` este folosită pentru simplitate, dar **are limitări** – de exemplu, nu elimină semnele de punctuație și tratează grupuri de spații succesive ca generând cuvinte goale. În practică, pentru a obține cuvinte reale dintr-o frază se preferă o expresie regulată ce selectează secvențe de litere: `text.match(/\p{L}+/gu)`[7][8], eventual urmat de `toLocaleLowerCase()` pentru uniformizare (în special dacă sunt incluse litere cu diacritice). Aceasta ar asigura că "Ana, are mere!!!" produce cuvintele ["ana","are","mere"] ignorând virgulă, spații multiple și semne de exclamare. În plus, am folosit pragul de 3 caractere ca exemplu; funcția se poate generaliza pentru orice lungime minimă dorim, trecând-o ca parametru.

5.2. **Var. B** — *wordsfilterB.js* (ES5, folosind `Array.filter` cu funcție clasică)

```
function filterWordsLongerThan3(text) {
  const words = text.split(' ');
  return words.filter(function(word) {
    return word.length >= 3;
  });
}

console.log(filterWordsLongerThan3("ana are mere și prune"));
// -> [ 'mere', 'prune' ]
```

Explicație: A doua variantă utilizează metoda built-in **`Array.filter`**. Această metodă parcurge array-ul `words` și aplică funcția de test furnizată (în cazul nostru, funcția anonimă `function(word){...}`) fiecărui element. Dacă pentru un element funcția returnează *true* (adică trece testul), elementul respectiv este inclus în array-ul nou pe care `filter` îl construiește și îl returnează. În cazul nostru, testul `word.length >= 3` asigură filtrarea dorită. Rezultatul este direct returnat de `filterWordsLongerThan3` (nu mai trebuie să gestionăm manual un array de rezultat).

Această abordare este **declarativă/funcțională**: îi spunem sistemului *ce* condiție să aplice (`word.length >= 3`), iar `Array.filter` se ocupă de logica de iterare intern. Din perspectivă de complexitate, rămâne O(n) timp, O(n) spațiu pentru noul array. Diferența majoră este în lizibilitate și concizie: varianta B este mai scurtă și expresivă – *"filtrează words păstrând doar elementele ce satisfac condiția"*. În ES5 trebuia să folosim o funcție clasică ca argument la `filter`.

5.3. **Var. C** — *wordsfilterC.js* (ES6+, folosind `filter` cu funcție *arrow*)

```
const filterWordsLongerThan3 = (text) => {
  const words = text.split(' ');
  return words.filter(word => word.length >= 3);
};

console.log(filterWordsLongerThan3("ana are mere și prune"));
// -> [ 'mere', 'prune' ]
```

Explicație: Această variantă este foarte apropiată de B, dar profită de sintaxa *arrow function*. În apelul `words.filter(word => word.length >= 3)`, am înlocuit `function(word){ return ... }` cu `(word) => ...`, ceea ce face codul și mai succint. Funcția `filterWordsLongerThan3` însăși este definită ca arrow și asignată unei constante.

Rezultatul comportamental este identic cu varianta B. Avantajul principal este **scrierea concisă** – codul devine aproape o traducere directă a cerinței: "*returnează cuvintele care au lungime >= 3*". Acest gen de ***formulare declarativă*** este unul din atuurile paradigmelor funcționale. Intern, `Array.filter` creează tot un nou array și nu modifică pe cel original[9], așa cum subliniază și documentația MDN: "metoda `filter()` creează o copie superficială a unei porțiuni din array-ul dat, filtrată astfel încât să conțină doar elementele care trec testul furnizat"[9].

5.4. **Var. D** — *wordsfilterD.js* (folosind `Array.reduce` pentru filtrare)

```
function filterWordsLongerThan3(text) {
  const words = text.split(' ');
  return words.reduce((acc, word) => {
    if (word.length >= 3) acc.push(word);
    return acc;
  }, []);
}

console.log(filterWordsLongerThan3("ana are mere și prune"));
// -> [ 'mere', 'prune' ]
```

Explicație: În această soluție folosim `Array.reduce` ca alternativă la `filter`. Reduce primește un acumulator (`acc`) inițial – aici un array gol `[]` – și iterează peste `words`, apelând funcția callback la fiecare element. Dacă elementul îndeplinește condiția, îl adăugăm (push) în acumulator. La final, `reduce` ne întoarce acumularea – adică tocmai array-ul de cuvinte filtrate. Practic, am reimplementat logica lui `filter` "manual" cu reduce.

Deși în acest caz nu există un avantaj major față de varianta C, exemplul demonstrează flexibilitatea lui `reduce` de a produce orice fel de rezultat prin iterare (nu doar sumă sau produs cum e folosit adesea). Codul este în continuare destul de declarativ (accentul e pe *ce facem* la fiecare pas).

Complexitatea este O(n) timp. Din punct de vedere al stilului, un avantaj minor este că nu mai necesită declarația separată a unei variabile intermediare `words` dacă am fi scris `text.split(' ').reduce(...)` direct – deși am preferat claritatea pas cu pas.

Notă: folosirea `reduce` aici este mai mult demonstrativă; în practică pentru filtrare directă e mai lizibil să folosim `filter`.

5.5. **Var. E** — *wordsfilterE.js* (imperativ modern, folosind `forEach`)

```
function filterWordsLongerThan3(text) {
  const words = text.split(' ');
  const result = [];
  words.forEach(word => {
    if (word.length >= 3) {
      result.push(word);
    }
  });
  return result;
}

console.log(filterWordsLongerThan3("ana are mere și prune"));
// -> [ 'mere', 'prune' ]
```

Explicație: Această variantă reia ideea imperativă a variantei A (iterare și construire manuală a rezultatului), dar utilizează metoda `Array.forEach` pentru a parcurge array-ul `words`. Metoda `forEach` execută funcția dată ca parametru (aici arrow function-ul `word => { ... }`) pe fiecare element, în ordine, fără a crea un array nou și fără a returna vreo valoare (returnează `undefined`). În interiorul callback-ului, aplicăm logica de filtrare: dacă lungimea e >=3, facem `result.push(word)`.

Practic, `forEach` ne scutește de a scrie explicit bucla `for`, dar rezultatul e similar cu varianta A: *mutație in-place* a array-ului `result`. Din perspectiva stilului, unii consideră că `forEach` combinat cu arrow produce un cod mai succint și mai clar decât un `for` clasic (nu mai avem indexare manuală), deși efectul este același.

În privința complexității, tot O(n) timp, O(n) spațiu.

Această metodă arată cum putem îmbina *paradigma imperativă* (efecte secundare, push într-un array) cu sintaxa modernă de *callback* pentru un compromis între claritate și control.

5.6. **Var. F** — *wordsfilterF.js* (variantă integrată cu CLI)

```
"use strict";    //vezi nota de subsol[1]

function filterWordsLongerThan3(text) {
  return text.split(' ').filter(w => w.length >= 3);
}
```

[1] *Consulta: APPENDIX 1 - Ce este și când folosim „use strict" în JavaScript.docx*

```
// Integrare CLI:
const inputText = process.argv.slice(2).join(' ');
if (!inputText) {
  console.error("Utilizare: node wordsfilterF.js <text de filtrat>");
  process.exit(1);
}
console.log(filterWordsLongerThan3(inputText));
```

Explicație: Ultima variantă demonstrează **integrarea cu linia de comandă (Node.js CLI)**. Am marcat scriptul cu `"use strict";` (bune practici) și am definit funcția `filterWordsLongerThan3` similar cu variantele anterioare (am ales intern să combin direct `split` cu `filter` ca în varianta C). Partea nouă este că preluăm argumentele din linia de comandă: `process.argv.slice(2)` ia toate argumentele furnizate după numele scriptului, pe care apoi le unim cu spațiu (`.join(' ')`) pentru a reconstitui fraza de intrare. Dacă nu s-a dat niciun argument (`inputText` e șirul vid), afișăm un mesaj de eroare privind utilizarea corectă și ieșim cu cod de eroare 1. Altfel, apelăm funcția de filtrare pe textul obținut și afișăm rezultatul (care va fi un array de cuvinte păstrate). Pentru a testa acest script, se poate rula:

```
$ node wordsfilterF.js "ana are mere și prune"
```

Rezultatul în consolă va fi `["mere","prune"]`.

Observați că, deoarece output-ul este un array, Node îl afișează sub forma standard `["elem1","elem2"]`. Într-un scenariu real, poate am formata rezultatul altfel (ex. să afișăm cuvintele separate prin virgulă), dar pentru simplitate folosim reprezentarea implicită.

Observații: Această variantă nu aduce o metodă algoritmică nouă, dar este utilă pentru a vedea cum putem lua input din afara codului hardcodat. Adăugarea suportului CLI face ca scriptul să poată fi folosit ca utilitar standalone. Am folosit tot metoda declarativă `filter` (similar cu var. C) – așadar această variantă moștenește caracterul imutabil al acelei abordări (creăm array nou pentru rezultat, nu modificăm array-ul inițial de cuvinte).

După cum se observă, toate variantele A–F produc același rezultat pentru aceeași intrare, dar diferă ca stil și tehnici folosite. Variantele B, C, D, F sunt funcționale (nu modifică array-ul original de cuvinte; B și C folosesc direct `filter`, D și F creează un nou array), pe când A și E au un caracter imperativ (construind manual rezultatul cu push). Ca performanță, toate au complexitate lineară în numărul de cuvinte. Diferențe minore apar la overhead: variantele funcționale creează array-uri noi implicit, A și E îl creează explicit – deci memoria suplimentară folosită e similară. Alegerea între ele depinde în principal de preferințele de cod și de contextul folosit: de exemplu, într-un cod de producție, var. C (arrow + filter) ar fi probabil cea mai "curată".

6. Calculul ariei si perimetrului (SquareDim A–E)

Cerința: Să se implementeze o funcție care, pentru o valoare dată reprezentând lungimea laturii unui pătrat, calculează **aria** și **perimetrul** pătratului. Practic, date de intrare un număr (latura), să se furnizeze două rezultate: aria = latura^2 și perimetrul = 4 × latura. Vom prezenta 5 variante de rezolvare a acestei sarcini. Deși formula de calcul este elementară, vom varia modul de returnare a rezultatelor și stilul de implementare (de la abordări imperative la utilizarea facilităților moderne).

6.1. **Var. A** — *squareDimA.js* (funcție imperativă clasică)

```
function squareProperties(side) {
  if (typeof side !== "number") {
    throw new TypeError("Argumentul trebuie să fie număr.");
  }
  const area = side * side;
  const perimeter = 4 * side;
```

```
  return [area, perimeter];
}

// Test
console.log(squareProperties(5));
// -> [25, 20]
```

Explicație: Funcția `squareProperties` din această variantă primește un parametru `side`. Mai întâi verificăm dacă este numeric, altfel aruncăm o eroare de tip (aceasta este o bună practică pentru robustețe). Apoi calculăm aria (`area = side * side`) și perimetrul (`perimeter = 4 * side`) și le **returnăm sub formă de array** cu două elemente: primul element este aria, al doilea perimetrul.

Am ales array pentru a păstra ambele valori într-o singură structură; alternativ, puteam returna un obiect sau putea afișa direct valorile. Varianta imperativă pune accent pe claritate: se folosesc variabile intermediare pentru a evidenția calculele, apoi se construiește rezultatul. **Mutația** are loc doar în sensul că definim două variabile locale (`area` și `perimeter`), dar nu modificăm niciun argument de intrare – deci nu există efecte secundare externe. Rezultatul fiind un array, se poate destructura la apel (de exemplu):

```
const [a, p] = squareProperties(5);.
```

Complexitatea este O(1) (operații aritmetice constante). Memoria suplimentară este neglijabilă (două variabile + array de 2 elemente).

6.2. **Var. B** — *squareDimB.js* (returnează obiect cu proprietăți)

```
function squareProperties(side) {
  if (typeof side !== "number") {
    throw new TypeError("Argumentul trebuie să fie număr.");
  }
  return {
    area: side * side,
    perimeter: 4 * side
  };
}

console.log(squareProperties(5));
// -> { area: 25, perimeter: 20 }
```

Explicație: A doua variantă realizează același calcul dar **returnează un obiect** cu două proprietăți denumite, în loc de array. Obiectul literal `{ area: side*side, perimeter: 4*side }` conține perechi cheie-valoare, astfel încât la apel putem accesa rezultatele după nume: ex. `let result = squareProperties(5); console.log(result.area, result.perimeter)`.

Avantajul acestei abordări este că **etichetează semnificația** fiecărei valori returnate, evitând confuzia de poziție (la array trebuia să ții minte că indexul 0 e aria și 1 e perimetrul). Codul este în continuare foarte simplu, folosind return direct al obiectului (fără variabile intermediare). Observați că am păstrat verificarea tipului pentru a arunca eroare dacă inputul nu e numeric. Din punct de vedere al paradigmei, tot imperativă este (calculăm și returnăm), dar oferă o structură de date mai bogată ca output.

Performanța e identică în esență cu var. A (diferențe minime la alocarea unui obiect vs array de dimensiuni mici). Alegerea între array și obiect ține de preferința de design: dacă avem doar două valori clar definite (aria, perimetrul), obiectul poate fi mai clar la utilizare.

6.3. **Var. C** — *squareDimC.js* (arrow function, sintaxă concisă)

```
const squareProperties = side => ({
  area: side * side,
  perimeter: 4 * side
```

```
});

console.log(squareProperties(5));
// -> { area: 25, perimeter: 20 }
```

Explicație: A treia variantă demonstrează folosirea unei **funcții arrow** cu sintaxă foarte concisă. Am definit `squareProperties` ca arrow function cu un singur parametru `side`. În corpul funcției, am pus direct un literal de obiect în paranteze rotunde `(...)` – aceasta este o tehnică necesară deoarece, în lipsa acoladelor (care ar indica un bloc de cod), o funcție arrow interpretează `{ ... }` ca bloc, nu ca obiect. Folosind `=> ({ area: ..., perimeter: ... })`, forțăm arrow function să evalueze și să returneze acel obiect.

Practic, această variantă este echivalentă ca rezultat cu varianta B, doar că nu am mai scris explicit `return` (returnul este implicit) și nici bloc de funcție. Pentru simplitate, am omis și verificarea de tip – în practică totuși, fiind o funcție foarte scurtă, am putea dori să păstrăm validația. Însă aici scopul e să evidențiem minimalismul: în doar două rânduri definim funcția și putem calcula valorile. **Paradigma** este clar funcțională: arrow function pură, fără efecte secundare. Ca stil, un astfel de one-liner este elegant, dar trebuie documentat sau numele proprietăților clar alese (ceea ce am făcut: `area` și `perimeter` sunt intuitive).

Notă de stil: Atât în var. B cât și C am ales să nu mutăm deloc starea, calculând imediat expresiile matematice în loc să folosim variabile. Acest lucru este posibil când expresiile sunt suficient de simple (ca aici). În situații mai complexe, definirea unor variabile intermediare poate crește lizibilitatea sau permite logarea valorilor intermediare, deci nu ezitați să le folosiți când e necesar, chiar și în funcții arrow (caz în care trebuie corp cu `{}` și `return` explicit).

6.4. **Var. D** — *squareDimD.js* (calcul multiplu folosind array methods)

```
function squareProperties(side) {
  if (typeof side !== "number") {
    throw new TypeError("Argumentul trebuie să fie număr.");
  }
  return [side, side * 4].map((val, idx) => {
    return idx === 0 ? val * side : val;
  });
}

console.log(squareProperties(5));
// -> [25, 20]
```

Explicație: Această variantă este mai *artistică* și folosește metode de array într-un mod inedit pentru a genera rezultatul. În loc să calculăm direct aria și perimetrul, am construit un array cu două elemente: `[side, side * 4]`. Acesta conține în prima poziție latura (side) și în a doua poziție perimetrul (4*side). Apoi aplic `.map(...)` asupra acestui array, transformându-l: pentru fiecare `val` și index `idx`, dacă indexul este 0 (primul element), înlocuim `val` cu `val * side` (practic latura * latura = aria); dacă indexul este 1, lăsăm `val` neschimbat (deja conține 4*side, adică perimetrul). Rezultatul final al lui `map` va fi un array `[side*side, side*4]`, adică `[aria, perimetru]`.

De fapt, această soluție calculează aria într pas separat, prin map. Pare cam complicat față de soluțiile anterioare, însă scopul este demonstrativ: arată că putem combina valori într-un array și apoi folosi `map` pentru a aplica formule diferite pe poziții diferite, pe baza indexului. Este totuși o abordare **mai puțin clară** și nu ar fi prima opțiune într-un mediu real – a fost aleasă aici pentru a sublinia un stil funcțional cu `map` și operator ternar.

De menționat că varianta D creează două array-uri: unul temporar `[side, side*4]` și unul final returnat de `map`. E ineficient față de calculul direct, dar pentru un singur număr costul e insignifiant. Totuși, ilustrează un principiu: uneori, ***programatorii pot abuza de metode funcționale chiar și***

acolo unde un calcul direct era mai simplu. E important să recunoaștem momentul potrivit pentru fiecare abordare.

6.5. **Var. E** — *squareDimE.js* (variantă CLI – citire din argumente)

```
"use strict";  //Nota de subsol[2]
function squareProperties(side) {
  const area = side * side;
  const perimeter = 4 * side;
  return { area, perimeter };
}

// Integrare CLI:
const arg = process.argv[2];
if (!arg) {
  console.error("Utilizare: node squareDimE.js <lungime_latura>");
  process.exit(1);
}
const side = Number(arg);
if (!Number.isFinite(side)) {
  console.error("Argument invalid. Vă rugăm furnizați un număr.");
  process.exit(1);
}
console.log(squareProperties(side));
```

Explicație: Varianta E combină calculul cu logica de citire a argumentelor din linia de comandă. Funcția `squareProperties` aici este o versiune simplificată (fără verificare internă, se presupune că primește număr) și returnează un obiect `{ area, perimeter }` (notă: am folosit *shorthand property names* – scriind `{ area, perimeter }` în loc de `{ area: area, perimeter: perimeter }`, valabil când variabilele se numesc la fel ca proprietățile dorite). Partea de sub „Integrare CLI” extrage al doilea argument din `process.argv` (primul argument real după numele scriptului) drept `arg`. Dacă nu există argument, afișăm modul de utilizare și oprim execuția cu cod 1. Dacă există, îl convertim la număr cu `Number(arg)` și verificăm cu `Number.isFinite` că rezultă un număr finit valid. Dacă conversia e nereușită (de ex. s-a dat un text ne-numeric), afișăm un mesaj de eroare. În final, apelăm `squareProperties(side)` și afișăm rezultatul (un obiect).

Pentru a folosi acest script, rularea ar arăta astfel:

```
$ node squareDimE.js 5
```

și programul ar afișa `{ area: 25, perimeter: 20 }`. Dacă nu furnizăm argument sau punem unul incorect, vom vedea mesajele de eroare corespunzătoare.

Observații: Această variantă evidențiază interacțiunea cu utilizatorul (printr-un parametru numeric așteptat). Am preferat să separăm clar responsabilitățile: funcția `squareProperties` face strict calculul matematic, iar codul din afara funcției se ocupă de input/output (citire, validare, afișare). **Aceasta este o bună practică – funcțiile ar trebui, ideal, să fie *pure* (fără I/O direct) pentru a fi ușor de testat și reutilizat, iar I/O să fie la periferia programului.**

În concluzie, exemplul **SquareDim** a demonstrat că și o problemă foarte simplă poate fi abordată în multiple feluri: de la cod explicit imperativ până la trucuri cu map și integrare cu sistemul. Variantele de returnare (array vs. obiect) ilustrează opțiuni de design – aici obiectul e probabil preferabil pentru claritate, mai ales dacă am extinde funcția să calculeze și diagonala, de exemplu. Important este că indiferent de stil, rezultatul e același și cunoaștem costurile/beneficiile fiecărei metode.

---

[2] *Consulta: APPENDIX 1 - Ce este și când folosim „use strict” în JavaScript.docx*

7. Formatarea numerelor (Formatter A–F)

Cerința: Să se implementeze o funcție care primește un număr întreg și o lungime totală și **formatează numărul ca string, suplimentand in prefix cu zerouri** dacă numărul are mai puține cifre decât lungimea cerută. Practic, comportamentul e similar cu formatul %0Nd din C (de ex., pentru N=4, numărul 42 devine "0042"). Vom realiza acest lucru în 6 variante, evidențiind soluții iterative, recursive și metode moderne (`String.padStart`).

Exemplu: Pentru numărul 7 și lungimea 3, rezultatul așteptat este "007". Pentru numărul 1234 și lungimea 2, rezultatul este "1234" (deoarece are deja mai mult de 2 cifre, nu se taie, ci se lasă întreg).

7.1. **Var. A** — *formatterA.js* (buclă `for` și conversie manuală)

```
function formatNumber(num, width) {
  if (!Number.isInteger(num) || !Number.isInteger(width)) {
    throw new TypeError("Parametri trebuie să fie întregi.");
  }
  let numStr = num.toString();
  for (let i = numStr.length; i < width; i++) {
    numStr = "0" + numStr;
  }
  return numStr;
}

console.log(formatNumber(7, 3));    // -> "007"
console.log(formatNumber(1234, 2)); // -> "1234"
```

Explicație: Funcția `formatNumber(num, width)` convertește mai întâi numărul în string (`num.toString()`) și stochează rezultatul în `numStr`. Apoi folosește o buclă `for` pentru a adăuga caractere '0' la stânga atât timp cât lungimea actuală a stringului `numStr` este mai mică decât `width` dorit. Dacă `num` are deja mai multe cifre decât `width`, bucla nu va itera deloc și `numStr` va rămâne neschimbat (de ex. cazul 1234 cu width 2). Returnăm stringul obținut. Am inclus o verificare a tipurilor: dacă `num` sau `width` nu sunt întregi, ***aruncăm eroare*** – aici e mai strict decât cerința, dar ajută la evitarea unor cazuri bizare (ex: număr zecimal, width negativ etc.).

Această variantă este **imperativă**, foarte clară ca algoritm adica spunem cumva: "*cât timp stringul e prea scurt, adaugă '0'*".

Complexitatea e O(n) unde *n = width - numStr.length* în cel mai rău caz (numărul de zerouri adăugate). Dacă `width` este mult mai mare decât numărul de cifre efective, bucla poate consuma timp proporțional. Dar în majoritatea cazurilor practice, width va fi rezonabil (de ex. formatare la 2-6 cifre pentru afișare).

Aceasta este metoda tradițională, similară cu exemplul de pe MDN care arată adăugarea de zerouri folosind buclă for[10].

7.2. **Var. B** — *formatterB.js* (versiune recursivă)

```
function formatNumber(num, width) {
  if (!Number.isInteger(num) || !Number.isInteger(width)) {
    throw new TypeError("Parametri trebuie să fie întregi.");
  }
  const numStr = num.toString();
  if (numStr.length >= width) {
    return numStr;
  }
  // Apel recursiv pentru a prelua restul de zerouri necesare
  return formatNumber(0, width - numStr.length) + numStr;
}
```

```
console.log(formatNumber(7, 3));    // -> "007"
console.log(formatNumber(1234, 2)); // -> "1234"
```

Explicație: Varianta recursivă `formatNumber` funcționează pe principiul că dacă stringul este prea scurt, putem prefixa un '0' și apoi încerca din nou până ajungem la lungimea dorită. Implementarea de față face un truc: dacă lungimea actuală `numStr` este deja >= width, returnează `numStr` direct (cazul de bază, nimic de completat). Dacă nu, apelează `formatNumber(0, width - numStr.length)` și concatenează cu `numStr`. Să analizăm: apelul recursiv primește numărul 0 și ca lățime `width - numStr.length`. Ce face acesta? Va produce un șir de lungime `width - numStr.length` plin de zerouri (deoarece 0 convertit la string e "0", iar lățimea cerută fiind mai mare, recursia continuă până obține atâtea zerouri). Practic, dacă `numStr = "7"` și `width = 3`, atunci apelul recursiv devine `formatNumber(0, 2)` – acesta ar trebui să returneze "00". Când concatenăm "00" + "7", obținem "007". Dacă `numStr` are deja lungime >= width, ramura recursivă nu e executată și întoarcem direct `numStr`.

Această abordare este interesantă dar puțin forțată; am folosit recursivitatea indirect, sprijinindu-ne pe faptul că formatând numărul 0 la o anumită lungime obținem doar zerouri. Puteam la fel de bine să facem recursiv ceva gen: `return "0" + formatNumber(num, width-1)` – ceea ce ar fi fost o recursivitate directă alternativă. Oricum, varianta B ilustrează o soluție **recursivă**, care deși nu e necesară aici (problema se rezolvă mai simplu iterativ), servește didactic: se bazează pe împărțirea problemei în sub-probleme mai mici (completarea a `width - currentLength` zerouri, apoi concatenarea cu numărul actual). E corectă dar mai dificil de urmărit decât varianta A.

Complexitatea e tot O(n) (practic tot atâtea concatenări se fac, dar realizate în stiva de apeluri recursive). Trebuie atenție la valoarea lui `width`: dacă s-ar da un width uriaș (ex. 10000), recursivitatea ar putea adânci stiva și duce la depășirea limitei (stack overflow). În cazuri normale, nu vom folosi recursiv abordarea asta pentru că nu are un beneficiu real față de buclă, ba chiar e mai costisitoare (concatenări repetate de stringuri, care în JS creează obiecte noi la fiecare pas).

7.3. **Var. C** — *formatterC.js* (utilizând `String.padStart`)

```
function formatNumber(num, width) {
  if (!Number.isInteger(num) || !Number.isInteger(width)) {
    throw new TypeError("Parametri trebuie să fie întregi.");
  }
  const numStr = num.toString();
  return numStr.padStart(width, "0");
}

console.log(formatNumber(7, 3));    // -> "007"
console.log(formatNumber(1234, 2)); // -> "1234"
```

Explicație: A treia variantă profită de funcționalitatea introdusă în ES2017 – metoda **`String.padStart(targetLength, padString)`**. Aceasta **extinde un string la lungimea dorită prin adăugarea unui *padding* la începutul șirului**, folosind un caracter sau șir specificat (aici folosim `"0"`). Practic, apelul `numStr.padStart(width, "0")` returnează un nou string de lungime `width` care conține stringul original `numStr` aliniat la dreapta și completat la stânga cu '0' ori de câte ori e nevoie[11]. Dacă `numStr` e deja de lungime >= width, `padStart` îl va returna nemodificat (nu taie din el, doar nu adaugă nimic[12]). Astfel, exemplul corespunde exact cerinței.

Această variantă este **cea mai concisă și expresivă**: practic delegăm toată munca lui `padStart`, care este optimizată în engine. Documentația MDN descrie `padStart` ca fiind echivalentul unui *polyfill* pentru operația de umplere cu caractere la stânga[13]. În exemplul dat acolo, se arată cum poate fi implementat cu `toString().padStart` un comportament similar cu `printf("%0*d")` din C[14], exact ce facem și noi. Avantajul major este lizibilitatea: oricine citește codul `.padStart(width, "0")` înțelege imediat intenția (*dacă cunoaște API-ul*).

Ca performanță, intern va fi similar cu o buclă implementată nativ, posibil chiar mai eficient decât bucla în JavaScript pur.

Aceasta este varianta preferată în ES7+ și este un exemplu perfect al paradigmei **declarative**: spunem direct *"păzește stringul la stânga cu 0 până la lungimea width"*, fără a specifica cum să itereze.

7.4. **Var. D** — *formatterD.js* (truc cu concatenare și `slice`)

```
function formatNumber(num, width) {
  if (!Number.isInteger(num) || !Number.isInteger(width)) {
    throw new TypeError("Parametri trebuie să fie întregi.");
  }
  const numStr = num.toString();
  const zeroes = "0".repeat(width); // un șir de '0' repetat de 'width' ori
  const padded = zeroes + numStr;
  return padded.slice(-Math.max(width, numStr.length));
}

console.log(formatNumber(7, 3));    // -> "007"
console.log(formatNumber(1234, 2)); // -> "1234"
```

Explicație: În această variantă, folosim o tehnică clasică întâlnită adesea înainte de padStart: **concatenează un șir lung de zerouri în fața numărului, apoi taie la dimensiunea dorită** din partea dreaptă. Mai precis, generăm un string `zeroes` care conține `width` de zero-uri (folosind metoda `String.repeat(n)` din ES6). Apoi concatenăm `zeroes + numStr`. Dacă de exemplu `numStr="7"` și `width=3`, atunci `zeroes="000"` și `padded="0007"`. Dacă `numStr="1234"` și `width=2`, `zeroes="00"`, `padded="001234"`. Următorul pas: returnăm `padded.slice(...)`. Folosim `slice` cu indice negativ pentru a extrage ultimele N caractere. Noi vrem ca în rezultatul final să păstrăm fie `width` caractere, fie toate câte are numărul dacă acesta are mai multe cifre decât width. De aceea am luat `slice(-Math.max(width, numStr.length))`:

- **Dacă numărul are mai puține cifre decât width**, `Math.max(width, numStr.length)` va fi `width`, deci vom face `slice(-width)`, adică ultimele `width` caractere, care vor include toate cifrele numărului și destui de mulți de '0' în față cât să aibă lungimea width (practic exact formatat corect).
- **Dacă numărul are deja mai multe cifre decât width**, atunci `Math.max(width, numStr.length)` e `numStr.length`. Vom face `slice(-numStr.length)`, adică practic vom obține întreg stringul original al numărului, deoarece `padded` conține `zeroes + numStr` și în cazul acesta `numStr` e mai lung decât `zeroes`. De exemplu la "001234", slice la -6 (numStr.length=4, width=2 deci max=4) dă "001234"? Aici trebuie atenție: dacă numărul e mai lung decât width, noi vrem rezultatul final să fie numStr intact (fără tăieri). Dacă `width < numStr.length`, `zeroes + numStr` are lungime = width + numStr.length, slice(-numStr.length) va extrage exact numStr plus câteva zerouri? Să verificăm: pentru numStr="1234"(len 4), width=2, padded="00"+"1234"="001234" (len 6). slice(-4) ia ultimele 4 caractere din "001234", ceea ce este "1234" – exact numStr, deci e corect. În schimb, dacă am face slice(-width) (adică -2), ar fi fost "34", ceea ce era greșit. De aceea am folosit max.

Această metodă e eficientă și ușor de implementat. Unii preferă o variantă și mai scurtă, fără `repeat`: de exemplu `('0000000000' + numStr).slice(-width)` presupunând că width e sub 10, sau generând un șir de 64 de zerouri și apoi folosind slice(-width) pentru width până la 64 etc. Dar cu `repeat` putem crea exact câte trebuie. Ca stil, este o soluție **hibridă**: nu e complet declarativă ca padStart, dar nici manuală ca bucla – e mai mult un idiom/truc. Din fericire, padStart a făcut-o redundantă, însă e util de cunoscut, mai ales pentru medii vechi unde padStart nu există (acesta ar fi un polyfill conceptual).

Complexitatea: generarea șirului de zerouri e O(n) (unde n = width) și concatenarea + slice tot O(n) combinat. Deci tot linear. Un mic surplus e că mereu generăm width zerouri și concatenăm, chiar dacă numărul e mare și nu era nevoie – de exemplu pentru 6 cifre și width 2 am generat tot "00" redundante. Dar costul e minor.

7.5. **Var. E** — *formatterE.js* (versiune folosind `substr` iterativ)

```
function formatNumber(num, width) {
  if (!Number.isInteger(num) || !Number.isInteger(width)) {
    throw new TypeError("Parametri trebuie să fie întregi.");
  }
  let result = num.toString();
  while (result.length < width) {
    result = "0" + result;
  }
  // dacă numărul are mai multe cifre decât width, îl lăsăm întreg:
  return result.length > width ? result.substr(0) : result;
}

console.log(formatNumber(7, 3));    // -> "007"
console.log(formatNumber(1234, 2)); // -> "1234"
```

Explicație: Această variantă combină elemente din cele de mai sus. Folosim o buclă `while` (similară în funcționalitate cu for-ul din var. A) pentru a adăuga '0' în față până ce lungimea atinge width. Am inclus la final o linie care, teoretic, dacă lungimea e > width, face `result.substr(0)` – care e practic string-ul întreg; acea condiție e redundantă în logica noastră (result nu va fi > width decât dacă numărul inițial era > width, situație în care oricum nu am adăugat nimic, deci result e chiar numStr original). Am pus-o doar pentru a evidenția că nu tăiem numărul dacă e mai lung – returnăm tot ce avem. `substr(0)` e doar o modalitate de a clona stringul (dar aici nu era necesară clonarea, îl puteam returna direct). Aceasta variantă nu aduce ceva conceptual nou: este practic o reimplementare a A folosind `while`. Am inclus-o totuși ca a cincea variantă pentru a arăta că există multiple moduri de a scrie aceeași logică imperativă (for vs while aici e doar o preferință sintactică). În plus, `substr` este o metodă de string mai veche (înlocuită de `slice` de obicei, dar încă funcțională) pe care am ilustrat-o. `result.substr(0)` returnează subșirul lui result de la indexul 0 încolo – adică întregul string. Această linie, așa cum am menționat, nu schimbă nimic; dacă am fi vrut să limităm lungimea maximă, puteam folosi `result.substr(-width)` sau ceva similar, dar nu era cazul aici (noi vrem lungime minimă, nu maxima).

Observații: Dintre toate variantele, probabil C (padStart) și D (trucul cu zeros + slice) sunt cele mai elegante. Varianta E (și A) sunt explicite și ușor de înțeles pentru începători, de aceea apar adesea în cursuri introductive. Varianta B (recursivă) este cea mai puțin practică, dar am inclus-o pentru completitudine și pentru a arăta că uneori se poate rezolva și recursiv, deși nu e necesar.

7.6. **Var. F** — *formatterF.js* (variantă CLI cu argumente)

```
"use strict";
function formatNumber(num, width) {
  const numStr = num.toString();
  return numStr.padStart(width, "0");
}

// Argumente CLI: <numar> <latime>
const [,, numArg, widthArg] = process.argv;
if (numArg === undefined || widthArg === undefined) {
  console.error("Utilizare: node formatterF.js <numar_int> <latime>");
  process.exit(1);
}
const num = Number(numArg), width = Number(widthArg);
if (!Number.isInteger(num) || !Number.isInteger(width)) {
```

```
  console.error("Ambele argumente trebuie să fie numere întregi.");
  process.exit(1);
}
console.log(formatNumber(num, width));
```

Explicație: Ultima variantă integrează funcționalitatea într-un script CLI care ia două argumente: primul este numărul, al doilea este lățimea dorită. Funcția `formatNumber` este implementată concis, folosind `padStart` (varianta C, practic). Codul principal extrage `numArg` și `widthArg` din `process.argv` folosind destructurare de array (ignorăm primele două elemente care sunt node și calea scriptului). Dacă nu sunt furnizate ambele argumente, afișăm un mesaj de utilizare și ieșim. Apoi convertim argumentele la numere și verificăm că sunt întregi (aici am permis și numere negative ca input – ele vor fi formattate cu minusul în față și zerouri după minus dacă e cazul; verificarea de întregi trece și numerele negative). Dacă ceva nu e conform, afișăm eroare. În final, apelăm `formatNumber(num, width)` și afișăm rezultatul format.

Un exemplu de rulare:

```
$ node formatterF.js 42 5
```

Ar trebui să tipărească `00042`. Dacă s-ar apela cu argument neîntreg (ex "3.14" sau "abc"), se va prinde la validație și va da mesaj de eroare.

Observații: Această variantă face ușor de testat utilitarul din terminal și demonstrează cum outputul nostru este direct un string (fără decorări suplimentare). În cazul în care `num` are mai multe cifre decât `width`, funcția noastră returnează `num` ca string întocmai – deci nu "taie" numărul, ceea ce e bine. Dacă cineva ar dori un comportament diferit (de ex. să ia doar ultimele `width` cifre ale unui număr mare), ar trebui schimbată implementarea (ex: în var. D dacă am folosi mereu slice(-width) fără max, ar tăia; sau aici am putea pune `numStr.slice(-width)` dacă dorim asta). Dar conform cerinței tipice, nu vom trunchia numărul.

În concluzie, **Formatter** a ilustrat mai multe moduri de a adăuga caractere de **padding** unui string reprezentând un număr. Am văzut o evoluție: de la buclă manuală, la recursiv, la funcționalități de limbaj mai avansate precum `padStart`. În practică, când avem acces la o versiune modernă de JS, `.padStart` este soluția recomandată pentru claritate și concizie[15]. Totuși, cunoașterea metodelor manuale ne ajută să înțelegem ce face padStart "sub capotă" și să ne descurcăm în medii unde poate nu e disponibil (sau când lucrăm cu polifill-uri).

8. Înmulțirea elementelor unui array (MapMul A–D)

Cerința: Se dă un array de numere și un factor (număr). Să se returneze un **nou array** în care fiecare element din array-ul inițial este înmulțit cu factorul dat. Practic, o operație de *map* de înmulțire scalară. Vom parcurge 4 variante de rezolvare.

Exemplu: Pentru array-ul `[1, 2, 3]` și factorul `5`, rezultatul așteptat este `[5, 10, 15]`. Pentru array-ul `[-2, 0, 7]` cu factorul `3`, rezultatul este `[-6, 0, 21]`.

8.1. **Var. A** — *mapmulA.js* (buclă `for` imperativă)

```
function multiplyArray(arr, factor) {
  if (!Array.isArray(arr) || typeof factor !== "number") {
    throw new TypeError("Argumente invalide");
  }
  const result = new Array(arr.length);
  for (let i = 0; i < arr.length; i++) {
    result[i] = arr[i] * factor;
  }
```

```
  return result;
}

console.log(multiplyArray([1, 2, 3], 5));
// -> [5, 10, 15]
```

Explicație: Funcția `multiplyArray(arr, factor)` creează mai întâi un array `result` de aceeași lungime cu `arr` (folosim `new Array(arr.length)` pentru a prealoca array-ul rezultat cu dimensiunea corectă; altfel, puteam porni cu `[]` și face `push`, dar prealocarea are un mic avantaj de performanță și claritate că știm mărimea finală). Apoi iterăm de la 0 la `arr.length - 1`, calculând `arr[i] * factor` și stocând în `result[i]`. Astfel, la final `result` conține elementele dorite.

Această soluție este **imperativă** și mutativă asupra array-ului rezultat (umplem array-ul după ce l-am creat). Array-ul de intrare `arr` rămâne nemodificat. Complexitatea este O(n). Avantajul acestei abordări este controlul complet: putem eventual să sarim elemente, să terminăm mai devreme etc. – dar în acest caz nu e nevoie. Este varianta cea mai directă și probabil cea mai performantă brut (dacă factorul de constantă contează, bucla `for` clasică e foarte rapidă în JS).

Notă: am validat intrarea – dacă `arr` nu e array sau factor nu e număr, aruncăm eroare. În plus, dacă `arr` conține elemente care nu sunt numere, rezultatul va fi `NaN` pe pozițiile respective (datorită aritmeticii JS – exemplu: `undefined * 2` devine `NaN`). Nu am tratat explicit asta, asumând date corecte în array (sau se poate lăsa comportamentul implicit).

8.2. **Var. B** — *mapmulB.js* (`Array.map` cu funcție clasică)

```
function multiplyArray(arr, factor) {
  if (!Array.isArray(arr) || typeof factor !== "number") {
    throw new TypeError("Argumente invalide");
  }
  return arr.map(function(element) {
    return element * factor;
  });
}

console.log(multiplyArray([1, 2, 3], 5));
// -> [5, 10, 15]
```

Explicație: Aici folosim direct metoda **`Array.map`** pentru a genera array-ul transformat. `arr.map(func)` va construi un nou array aplicând funcția `func` fiecărui element din `arr`. Noi furnizăm o funcție anonimă clasică care primește `element` (și ar putea primi și index, dar nu îl folosim) și returnează `element * factor`. `factor` este luat din scope-ul părinte al funcției (închidere lexicală). Rezultatul lui `map` este array-ul cu elementele înmulțite.

Această variantă este *declarativă*: nu mai scriem noi bucla, ci specificăm doar transformarea fiecărui element. Este tot O(n) timp, O(n) memorie (creează array nou). Diferența majoră e concizia și claritatea: practic citind codul e limpede "mapează array-ul înmulțind fiecare element cu factor". Documentația MDN spune: "metoda `map()` creează un array nou populat cu rezultatul apelării unei funcții furnizate pe fiecare element al array-ului apelant"[16] – exact ce facem noi.

8.3. **Var. C** — *mapmulC.js* (`Array.map` cu funcție *arrow*)

```
const multiplyArray = (arr, factor) => arr.map(x => x * factor);

console.log(multiplyArray([1, 2, 3], 5));
// -> [5, 10, 15]
```

Explicație: Am condensat la maximum soluția folosind o **arrow function** atât ca funcție externă `multiplyArray`, cât și ca callback intern pentru `map`. Am renunțat la verificările de tip pentru a păstra exemplul scurt – atenție, în practică e bine să le avem, dar aici considerăm

input valid. Practic, definiția `multiplyArray = (arr, factor) => arr.map(x => x * factor)` face totul dintr-o linie: returnează direct rezultatul lui `arr.map(...)`. Callback-ul la `map` este `x => x * factor`.

Observați că `factor` este capturat din scope-ul parinte (al funcției arrow exterioare). Această variantă este echivalentă ca efect cu B, doar mai concisă. E foarte ușor de citit: "*pentru fiecare x din array înmulțește-l cu factor*". Este exemplul perfect de folosire a unei HOF (*higher-order function*) cu arrow pentru transformare.

Stilistic, acesta este adesea modul preferat în codul modern JS când nu sunt necesare tratamente speciale: e scurt și la obiect. E bine totuși să nu abuzăm de concizie în detrimentul clarității – în cazul de față e clar, dar dacă ar fi o transformare mai complicată poate ar merita desfășurat pe mai multe linii.

8.4. Var. D — *mapmulD.js* (folosind `Array.reduce`)

```
function multiplyArray(arr, factor) {
  if (!Array.isArray(arr) || typeof factor !== "number") {
    throw new TypeError("Argumente invalide");
  }
  return arr.reduce((acc, element) => {
    acc.push(element * factor);
    return acc;
  }, []);
}

console.log(multiplyArray([1, 2, 3], 5));
// -> [5, 10, 15]
```

Explicație: Ultima variantă arată o rezolvare folosind `Array.reduce`, ceea ce ne permite să *construim manual* array-ul rezultat într-o manieră funcțională. Inițializăm acumulantul `acc` ca array gol `[]`. La fiecare element al lui `arr`, funcția de reducere înmulțește elementul cu factor și îl adaugă (`push`) la array-ul acumulat, apoi returnează acumulantul. La final, `reduce` ne furnizează acumulantul final – care este chiar array-ul cu elementele înmulțite.

Acesta este practic echivalent ca logică cu varianta A (bucla imperativă) dar scris într-un stil declarativ cu `reduce`. Unii ar putea argumenta că e mai puțin eficienta decât varianta map, deoarece face push succesiv (dar complexitatea tot O(n) e, constanta e similara). Este mai mult pentru a demonstra versatilitatea lui `reduce`. Codul este relativ clar: *"reduce array-ul la un nou array acc, adăugând element*factor"*. Rețineți că folosim arrow function în callback-ul lui reduce.

Notă: Puteam folosi direct și `flatMap` într-un scenariu mai complex sau `forEach` (imperativ callback) – de exemplu, variantă alternativă cu forEach:

```
const result = [];
arr.forEach(el => result.push(el * factor));
return result;
```

Acesta ar fi analogul lui A scris cu callback, dar noi am preferat reduce pentru a rămâne în paradigmă funcțională. În general însă, pentru probleme de transformare 1-la-1 cum e aici, `map` este instrumentul dedicat și cel mai lizibil, deci variantelor B/C li se acordă prioritate în codul idiomatic.

Pentru **MapMul**, concluzia este similară cu la filtrare: varianta cu `Array.map` este cea mai concisă și expresivă, iar cele cu buclă sau reduce sunt educative. Dintre acestea, bucla clasică poate fi preferată doar în situații unde chiar contează fiecare fragment de performanță (rareori în practică), iar reduce e mai mult o demonstrație – folosirea reduce pentru a simula map sau filter este posibilă (reduce e foarte general), dar nu e necesară când avem direct map și filter disponibile.

9. Generarea unui acrostih (Acrostih A–H)

Cerința: Să se scrie un program care, dată o listă de cuvinte (de exemplu, versurile unui poem sau elementele unui șir de stringuri), construiește **acrostihul** asociat – adică formează un nou cuvânt concatenând prima literă a fiecărui cuvânt din listă, în ordinea dată. Practic, dacă avem input cuvintele

$$ "Mere", "Ana", "Are"$$

acrostihul ar fi "**MAA**" (prima literă din "Mere" + prima literă din "Ana" + prima literă din "Are"). Vom realiza 8 variante, explorând diferite modalități de obținere a primei litere și de iterare prin listă.

Exemplu: Input: `["rosu", "orange", "siena", "ultramarin", "alb"]`, acrostihul ar fi `"rosa u"`? Hmm, luând prima literă din fiecare: "rosu"[0]="r", "orange"[0]="o", "siena"[0]="s", "ultramarin"[0]="u", "alb"[0]="a" – deci "rosua". (Acesta pare un cuvânt, dar dacă nu dă ceva coerent, nu contează – e practic doar un acronim de litere).

Vom testa pe alte exemple de asemenea. Poate un exemplu mai clar:

$$"Soare", "Arde", "Luna", "Apare"$$

ar da "SALA".

9.1. **Var. A** — *acrostihA.js* (`for` clasic cu indexare)

```
function acrostic(words) {
  if (!Array.isArray(words)) throw new TypeError("Se așteaptă un array de stringuri.");
  let result = "";
  for (let i = 0; i < words.length; i++) {
    const word = words[i];
    if (typeof word === "string" && word.length > 0) {
      result += word[0];
    }
  }
  return result;
}

console.log(acrostic(["Mere", "Ana", "Are"]));
// -> "MAA"
```

Explicație: Această variantă parcurge lista de cuvinte cu un index numeric de la 0 la length-1. Pentru fiecare cuvânt, verificăm că este un string nevid (dacă evidențele cerinței spun că toate sunt cuvinte, s-ar putea omite, dar punem o verificare totuși). Apoi extragem prima literă cu sintaxa `word[0]` și o **concatenăm** la stringul `result`. Variabila `result` este inițial șirul vid `""` și la final conține acrostihul complet.

Observații: folosim `word[0]` în loc de `word.charAt(0)` – ambele funcționează la fel pentru a lua caracterul de la index. Sintaxa de indexare directă e mai modernă și suportă returnarea `undefined` dacă indexul e în afara limitelor (charAt ar întoarce "" șir vid în acel caz). În cazul nostru, ne asigurăm că `word.length > 0` înainte de acces.

Această variantă este imperativă, cu mutație pe variabila `result` la fiecare pas. Complexitate O(n) timp, O(1) spațiu extra (exceptând stringul final care are lungime n, dar asta e outputul dorit).

9.2. **Var. B** — *acrostihB.js* (`for...of` loop)

```
function acrostic(words) {
  if (!Array.isArray(words)) throw new TypeError("Se așteaptă un array de stringuri.");
  let result = "";
  for (const word of words) {
    if (typeof word === "string" && word.length > 0) {
```

```
      result += word[0];
    }
  }
  return result;
}

console.log(acrostic(["Mere", "Ana", "Are"]));
// -> "MAA"
```

Explicație: În loc să iterăm cu index, folosim instrucțiunea modernă **`for...of`** care ne dă direct elementele din array unul câte unul. Variabila `word` în interior ia pe rând valorile din `words`. Apoi logica e aceeași: dacă `word` este string nevid, concatenăm primul caracter la `result`. Rezultatul final este returnat.

Această variantă este puțin mai concisă decât A, evitând accesarea array-ului cu indice. De asemenea, scapă de nevoia de a declara explicit `word = words[i]`. În rest, comportamentul e identic.

Multe preferințe stilistice în JS merg spre `for...of` pentru a parcurge array-uri atunci când nu ai nevoie de index. Este considerat mai sigur (nu riști off-by-one sau index out of range, deși tot trebuie să verifici tipul elementelor dacă ai dubii). În cazul nostru, am menținut verificarea tipului elementului. Dacă ar exista elemente nesting (de ex. numere), acestea ar fi pur și simplu ignorate în acrostih (nu ar contribui litere).

9.3. **Var. C** — *acrostihC.js* (buclă `while`)

```
function acrostic(words) {
  if (!Array.isArray(words)) throw new TypeError("Se așteaptă un array de stringuri.");
  let result = "";
  let i = 0;
  while (i < words.length) {
    const word = words[i];
    if (typeof word === "string" && word.length > 0) {
      result += word.charAt(0);
    }
    i++;
  }
  return result;
}

console.log(acrostic(["Mere", "Ana", "Are"]));
// -> "MAA"
```

Explicație: Același proces, dar folosind un `while` în loc de `for`. Inițializăm `i=0`. Cât timp `i` este mai mic decât lungimea array-ului, preluăm elementul `words[i]`, verificăm și concatenăm prima literă (aici am folosit alternativ `word.charAt(0)` – face același lucru ca `word[0]` pentru string; de dragul diversității l-am utilizat, deși preferința modernă e spre indexare). Apoi incrementăm `i`. Rezultatul e construit la fel și returnat.

Acesta este modul clasic de a folosi un `while`. Codul e mai lung decât varianta cu `for` pentru că trebuie să gestionăm incrementarea manual și inițializarea separat, dar face același lucru.

Stilistic, `while` e rar necesar pentru parcurgerea secvențială cunoscută (când știi limitele de parcurs). Dar e instructiv să arătăm că se poate. Performanța e aceeași, diferențele țin de lizibilitate/risc de bug (de ex. ușor să uiți `i++` și intri în loop infinit; cu for e mai compact).

9.4. **Var. D** — *acrostihD.js* (`Array.forEach`)

```
function acrostic(words) {
  if (!Array.isArray(words)) throw new TypeError("Se așteaptă un array de stringuri.");
  let result = "";
  words.forEach(word => {
```

```
    if (typeof word === "string" && word.length > 0) {
      result += word[0];
    }
  });
  return result;
}

console.log(acrostic(["Mere", "Ana", "Are"]));
// -> "MAA"
```

Explicație: Folosim metoda de iterație **`Array.forEach`**, care apelează funcția de callback pentru fiecare element din array. Funcția noastră callback (arrow) primește direct `word` ca parametru (forEach mai oferă și index și întregul array ca param. opționali, dar nu avem nevoie). În interior, logică identică: dacă e string nevid, `result += word[0]`. `forEach` nu returnează nimic (returnează `undefined`), de aceea construim într-o variabilă externă `result` definită anterior.

Aceasta este tot o abordare imperativă, deși scrisă cu callback. E un stil ceva mai declarativ (nu manual), deși conceptul de *side effect* rămâne (modificăm `result` din afara callback-ului).

Ca efect, forEach produce același rezultat. Unii ar critica folosirea forEach în loc de reduce/map atunci când vrei să obții ceva (fiindcă forEach e folosit mai mult pentru efecte secundare). În cazul nostru, generarea unui string e echivalent cu un efect secundar, deci e ok. Alternative ceva mai funcționale vin în variantele următoare.

9.5. **Var. E** — *acrostihE.js* (`Array.map` + `join`)

```
function acrostic(words) {
  if (!Array.isArray(words)) throw new TypeError("Se așteaptă un array de stringuri.");
  return words.map(word => {
    if (typeof word === "string" && word.length > 0) return word[0];
    return "";
  }).join("");
}

console.log(acrostic(["Mere", "Ana", "Are"]));
// -> "MAA"
```

Explicație: Această variantă este mai funcțională: folosim `Array.map` pentru a transforma lista de cuvinte într-o listă de litere (prima literă a fiecărui cuvânt). Apoi folosim `Array.join("")` ca să concatenăm acele litere într-un string final. Să detaliem: `words.map(word => { ... })` – pentru fiecare element: dacă este string nevid, returnăm `word[0]`; altfel returnăm `""` (șirul vid). Astfel, obținem un array, de aceeași lungime ca `words`, conținând fie litera initială dacă cuvântul era valid, fie șir vid dacă nu. De exemplu pentru `["Mere","", "Are"]` ar produce `["M", "", "A"]`. Apoi `.join("")` concatenează elementele array-ului cu separator șirul vid (deci le lipește direct unul după altul). Rezultatul este `"MA"`. În exemplul nostru fără elemente goale, e direct "MAA".

Avantajul acestei soluții este că evită să folosim o variabilă externă și efecte secundare. Construim totul prin compunere de funcții: map + join. Este idiomatic JS: de exemplu, MDN menționează adesea `array.map(...).join('')` ca soluție la probleme de transformare de array în string. Complexitatea e O(n) oricum. Un lucru de notat: dacă un element nu era string sau era "", codul nostru l-ar transforma în "", deci practic l-ar ignora în acrostih. Asta poate fi acceptabil sau nu, dar aici nu se specifică, așa că e ok.

9.6. **Var. F** — *acrostihF.js* (`Array.reduce`)

```
function acrostic(words) {
  if (!Array.isArray(words)) throw new TypeError("Se așteaptă un array de stringuri.");
  return words.reduce((acc, word) => {
```

```
    if (typeof word === "string" && word.length > 0) {
      return acc + word.charAt(0);
    }
    return acc;
  }, "");
}

console.log(acrostic(["Mere", "Ana", "Are"]));
// -> "MAA"
```

Explicație: Varianta cu `Array.reduce` acumulează direct într-un string `acc` pornind de la valoarea inițială `""`. Funcția de reducere concatenează la acumulant prima literă a `word` curent dacă e cazul (string nevid), altfel lasă acumulantul neschimbat. Apoi returnează acumulantul pentru următoarea iterație. Astfel, la final, reduce produce șirul complet.

Această tehnică este echivalentă logic cu varianta D (forEach) dar scrisă într-un mod funcțional (fără variabile externe, folosind return în fiecare pas). Este elegantă și clară: definim modul în care se combină un element cu acumulantul. În caz că un element nu contribuie (nu e string nevid), acumulantul e returnat nemodificat.

Din punct de vedere al eficienței, tot O(n), cu n concatenări. Ca și la variantă D, concatelarea repetată de stringuri în buclă poate fi puțin mai costisitoare decât un join la final (depinde de implementare; unele motoare optimizează concatenările succesive de stringuri scurte, transformându-le intern în logică de builder). Dar oricum n e mic în majoritatea cazurilor. În plus, varianta E cu join creează un array intermediar de litere, pe când reduce creează direct stringul fără array suplimentar. Deci reduce ar putea economisi memorie, cu costul unor concatenări repetate. Oricum, diferențele sunt minore.

9.7. **Var. G** — *acrostihG.js* (versiune recursivă)

```
function acrostic(words) {
  if (!Array.isArray(words)) throw new TypeError("Se așteaptă un array de stringuri.");
  if (words.length === 0) {
    return "";
  }
  const [first, ...rest] = words;
  const firstChar = (typeof first === "string" && first.length > 0) ? first[0] : "";
  return firstChar + acrostic(rest);
}

console.log(acrostic(["Mere", "Ana", "Are"]));
// -> "MAA"
```

Explicație: Aceasta variantă folosește recursivitatea pe lista de cuvinte. Dacă lista e goală (`words.length === 0`), acrostihul e stringul vid (cazul de bază). Altfel, *decompunem* lista în primul element `first` și restul `rest` (utilizând **destructurare**: `[first, ...rest] = words`). Aflăm prima literă a lui `first` dacă e string nevid, altfel e `""`. Notăm asta în `firstChar`. Apoi returnăm `firstChar + acrostic(rest)`. Astfel, funcția se apelează recursiv pe restul listei și concatenează litera curentă în față. Imaginează: pentru `["Mere","Ana","Are"]`: first="Mere", rest=["Ana","Are"], firstChar="M". Returnează "M" + acrostic(["Ana","Are"]). Acel apel va returna "AA". Deci obținem "MAA".

Această abordare este elegantă și profită de sintaxa ES6 (destructurare, rest parameter). Este însă mai greu de urmărit pentru un începător decât variantele iterative. De asemenea, recursivitatea are limita de stivă dacă lista e foarte lungă. În JavaScript, recursivitatea nu e optimizată (nu există TCO garantat), deci bucla e preferabilă pentru date mari. Dar pentru scop didactic, e o soluție validă.

9.8. **Var. H** — *acrostihH.js* (variantă CLI completă)

```
"use strict";
function acrostic(words) {
```

```
  return words.reduce((acc, word) => (typeof word === "string" && word[0]) ? acc +
word[0] : acc, "");
}

// Citire argumente din linia de comandă
// Exemplu de apel: node acrostihH.js "mere ana are"
const inputText = process.argv.slice(2).join(" ");
if (!inputText) {
  console.error("Utilizare: node acrostihH.js <sir de cuvinte>");
  process.exit(1);
}
const cuvinte = inputText.split(/\s+/);
console.log(acrostic(cuvinte));
```

Explicație: În această variantă finală, integrăm funcționalitatea într-un script Node care citește o frază completă ca argument din linia de comandă. Am combinat soluția anterioară reduce (varianta F) într-o singură linie arrow function pentru concizie: `reduce((acc, word) => (conditie) ? acc + primaLitera : acc, "")`. Observație: aici am scris `(typeof word === "string" && word[0]) ? ...` – aceasta are ca efect că dacă `word` nu e string sau e string gol, atunci `word[0]` e falsy (`undefined` sau `""`), deci condiția va fi falsă și nu se concatează nimic. Dacă `word` e string nevid, `word[0]` e un caracter (truthy) și se concatenează. E un mic artificiu pentru a scurta expresia, dar funcționează.

Codul CLI: Folosim `process.argv.slice(2).join(" ")` pentru a lua toate argumentele după numele scriptului și le unim într-un text unic (presupunem că inputul e dat ca un string cu spații, e mai simplu să-l reconstruim decât să cerem userului să pună ghilimele; dar în exemplu am pus ghilimele ca să intre tot într-un argument, altfel fără ghilimele process.argv ar avea ele separate; codul nostru însă pune spații la loc oricum). Dacă după join rezultă șir vid, înseamnă că nu s-a dat input – afișăm un mesaj de utilizare. Altfel, facem `inputText.split(/\s+/)` care împarte textul după spații (un regex simplu ce separă după unul sau mai multe spații, ca să ignore eventual spațiile multiple). Obținem array-ul de cuvinte `cuvinte`. Apoi apelăm `acrostic(cuvinte)` și afișăm rezultatul.

Astfel putem rula:

```
$ node acrostihH.js Mere Ana are
MAa
```

Observație: am dat trei argumente separate (Mere, Ana, are) fără ghilimele. `join(" ")` le va uni cu spațiu, deci `inputText` devine `"Mere Ana are"` (exact ca un singur argument ar fi fost "Mere Ana are"). split(/\s+/) dă ["Mere","Ana","are"]. Acrostih va da "MAa" (atenție, 'are' e scris cu litere mici, deci litera e 'a' mic – programul nu schimbă cazul). Dacă vrem acrostih cu toate literele majuscule, am fi putut normaliza inputul la un anumit case. Dar cerința nu specifică, deci lăsăm așa.

Observații: Varianta H evidențiază și problema literelor cu diacritice sau caractere multibyte – de exemplu, cuvântul "Șarpe" are [0] un caracter "Ș" (2 code units UTF-16). Dar JavaScript `word[0]` va returna acel caracter complet (nu un cod incorect) deoarece `for...of` sau accesarea directă de index tratează stringurile ca secvențe de unități UTF-16, ceea ce pentru literele românești funcționează (sunt în Basic Multilingual Plane, deci tot 1 code unit). Dar dacă ar fi emoji sau litere compuse, [0] ar putea prinde doar primul code unit dintr-un caracter compus. O soluție robustă ar fi să normalizăm și să folosim `Array.from(word)[0]` sau `word.codePointAt(0)` combinat cu fromCodePoint etc. Însă intrăm prea adânc – nu e necesar aici.

Prin aceste 8 variante, am ilustrat foarte multe elemente ale limbajului: bucle imperative diverse, metode de iterare funcționale (`map`, `reduce`, `forEach`), recursivitate, destructurare, regex, CLI etc. Soluția preferată pentru acrostih, într-un context real, probabil ar fi o combinație concisă gen variantă E sau F (folosind map+join sau reduce), eventual cu o simplă validare.

10. Filtrarea proprietăților unui obiect (Keys&Filters A–G)

Cerința: Dat un obiect care conține perechi cheie-valoare și un predicat de filtrare aplicat valorilor, să se obțină un **sub-obiect** care conține doar acele perechi care satisfac filtrul (sau, alternativ, lista cheilor care satisfac). Cu alte cuvinte, să extragem dintr-un "dicționar" doar anumite elemente pe baza valorii lor. Vom considera că predicatul este fixat (de exemplu "valoarea este un număr peste o anumită limită" sau "valoarea este de tip string care conține o subsecvență"). Vom prezenta 7 variante, evidențiind atât utilizarea ciclurilor, cât și a metodelor precum `Object.keys`, `Object.entries`, combinat cu `filter` și alte tehnici.

Exemplu de scenariu: Să filtrăm un obiect ce conține scoruri ale unor jucători, păstrând doar pe cei cu scor peste 50. Input: `{ Ana: 20, Bogdan: 75, Ion: 50, Mia: 120 }`. Așteptat: fie `{ Bogdan: 75, Mia: 120 }` (sub-obiect) dacă vrem obiect rezultat, fie `["Bogdan","Mia"]` dacă vrem lista cheilor. Vom varia output-ul între variante. Predicatul de filtrare: `valoare > 50`.

Vom implementa predicatul direct în cod pentru simplitate (nu îl dăm ca parametru separat în aceste exemple).

10.1. **Var. A** - *keysfilterA.js* (buclă `for...in`, construind obiect rezultat)

```
function filterObject(obj) {
  if (obj === null || typeof obj !== "object") {
    throw new TypeError("Argumentul trebuie să fie un obiect.");
  }
  const result = {};
  for (const key in obj) {
    if (Object.prototype.hasOwnProperty.call(obj, key)) {
      const value = obj[key];
      if (typeof value === "number" && value > 50) {
        result[key] = value;
      }
    }
  }
  return result;
}

console.log(filterObject({ Ana: 20, Bogdan: 75, Ion: 50, Mia: 120 }));
// -> { Bogdan: 75, Mia: 120 }
```

Explicație: Aici folosim bucla **`for...in`** pentru a itera cheile unui obiect. `for...in` va itera toate proprietățile enumerable ale obiectului, incluzând cele moștenite din prototip (dacă există). Ca bună practică, folosim `hasOwnProperty` pentru a ne asigura că filtrăm doar proprietățile "own" (directe ale obiectului, nu moștenite)[17]. În interior, preluăm `value = obj[key]`. Apoi aplicăm predicatul: dacă `value` este număr și > 50, atunci adăugăm această pereche la obiectul `result`. (Observație: testăm și tipul numeric, deși în exemplu știm scorurile sunt numere; am pus asta poate în eventualitatea altor tipuri de valori). După buclă, `result` conține doar proprietățile dorite, pe care îl returnăm.

Aceasta variantă produce un **sub-obiect** filtrat. E imperativă și mutativă/mutabila față de `result`. Complexitate O(n) în numărul de proprietăți (depinde de implementare, dar de obicei iterează n proprietăți).

10.2. **Var. B** - *keysfilterB.js* (folosind `Object.keys` și buclă)

```
function filterObject(obj) {
  if (obj === null || typeof obj !== "object") {
    throw new TypeError("Argumentul trebuie să fie un obiect.");
  }
  const keys = Object.keys(obj);
  const result = {};
```

```
  for (let i = 0; i < keys.length; i++) {
    const key = keys[i];
    const value = obj[key];
    if (typeof value === "number" && value > 50) {
      result[key] = value;
    }
  }
  return result;
}

console.log(filterObject({ Ana: 20, Bogdan: 75, Ion: 50, Mia: 120 }));
// -> { Bogdan: 75, Mia: 120 }
```

Explicație: Folosim metoda **`Object.keys(obj)`** pentru a obține un array cu cheile proprii ale obiectului (metoda exclude oricum prototipale)[18]. Apoi iterăm peste acel array `keys` cu un for numeric. Logica din interior e ca înainte: extragem value și dacă e >50 (predicatul), asignăm în result. Rezultatul e obiectul filtrat.

Diferența față de varianta A: aici `for...in` nu mai e folosit, deci nu mai trebuie `hasOwnProperty` (pentru că Object.keys deja garantează numai cheile "own"). De asemenea, se poate considera că e ușor mai curat pentru cine preferă lucrul cu array-uri.

E posibil ca `Object.keys` să copieze cheile într-un array, deci are un cost O(n) memorie suplimentară, dar la număr mic de proprietăți e neglijabil. Avantajul e că putem eventual aplica metode de array pe `keys` (vezi variante următoare).

10.3. **Var. C** — *keysfilterC.js* (folosind `Object.keys` și `filter`)

```
function filterKeys(obj) {
  if (obj === null || typeof obj !== "object") {
    throw new TypeError("Argumentul trebuie să fie un obiect.");
  }
  return Object.keys(obj).filter(key => {
    const value = obj[key];
    return (typeof value === "number" && value > 50);
  });
}

console.log(filterKeys({ Ana: 20, Bogdan: 75, Ion: 50, Mia: 120 }));
// -> ["Bogdan", "Mia"]
```

Explicație: Aici alegem să returnăm **lista cheilor** care îndeplinesc condiția, nu obiectul filtrat (pentru diversitate). Folosim `Object.keys(obj)` să obținem array-ul de chei. Apoi aplicăm direct metoda **`Array.filter`** pe acel array. În callback-ul lui `filter`, punem condiția: ținem cheia dacă valoarea asociată `obj[key]` este număr >50. Rezultatul lui filter va fi un array de chei care au trecut testul. Îl returnăm.

Exemplul dat confirmă: primim `["Bogdan","Mia"]`. Observație: Ion având exact 50 nu intră (50 nu e strict >50, deci rezultatul e ok).

Această abordare e (foarte) declarativă: "*filtrează cheile obiectului păstrându-le pe acelea pentru care valoarea corespunde condiției*". E concisă și ușor de extins cu alte condiții. De menționat: dacă obiectul are chei non-enumerable, Object.keys nu le include, deci e ok. Dacă are proprietăți simbol, nu sunt incluse, dar nu considerăm aici simboluri.

Complexitate: O(n) pentru keys + O(n) pentru filter => O(n) total.

10.4. **Var. D** - *keysfilterD.js* (folosind `Object.entries` și `filter` + `map`)

```
function filterKeys(obj) {
  if (obj === null || typeof obj !== "object") {
    throw new TypeError("Argumentul trebuie să fie un obiect.");
```

```
  }
  return Object.entries(obj)
    .filter(([key, value]) => typeof value === "number" && value > 50)
    .map(([key, value]) => key);
}

console.log(filterKeys({ Ana: 20, Bogdan: 75, Ion: 50, Mia: 120 }));
// -> ["Bogdan", "Mia"]
```

Explicație: Această variantă folosește **`Object.entries(obj)`** care returnează un array de perechi `[cheie, valoare]` pentru obiect[19]. Putem atunci aplica direct `filter` pe acest array de tupluri. Folosim destructurare în parametrii callback-ului: `([key, value]) => ...` ne permite să accesăm direct cheie și valoare. Condiția de filtrare este aceeași. Rezultatul lui filter va fi un array de perechi care au trecut testul. Apoi aplicăm `map` pe acest array pentru a extrage doar cheia din fiecare pereche filtrată (`map(([key, value]) => key)`). Obținem astfel un array de chei. Îl returnăm.

Acest lanț de operații este foarte lizibil fluent: "*ia entries, filtrează-le pe baza valorii, apoi extrage cheia*". E practic o formă de *comprehensiune*. Rezultatul e identic cu varianta C.

Dacă am fi vrut direct obiect rezultat (nu doar chei), am fi putut opri la filter și apoi transforma array-ul de entries într-un obiect la final, eventual cu `Object.fromEntries` (vezi variantă F).

10.5. **Var. E** - *keysfilterE.js* (folosind `Object.entries` și `reduce`)

```
function filterObject(obj) {
  if (obj === null || typeof obj !== "object") {
    throw new TypeError("Argumentul trebuie să fie un obiect.");
  }
  return Object.entries(obj).reduce((acc, [key, value]) => {
    if (typeof value === "number" && value > 50) {
      acc[key] = value;
    }
    return acc;
  }, {});
}

console.log(filterObject({ Ana: 20, Bogdan: 75, Ion: 50, Mia: 120 }));
// -> { Bogdan: 75, Mia: 120 }
```

Explicație: Folosim din nou `Object.entries`, dar de data aceasta aplicăm `reduce` pentru a construi direct obiectul filtrat (similar conceptului din var. A/B dar într-un mod funcțional). Inițializăm acumulantul ca `{}` (obiect gol). Pentru fiecare entry `[key, value]`, dacă valoarea trece condiția (număr >50), atunci adăugăm proprietatea respectivă la `acc`. Apoi returnăm `acc` pentru următoarea iterație. La final, `acc` este obiectul rezultat cu proprietățile adăugate. Îl returnăm.

Rezultatul e echivalent cu variantă A și B: sub-obiect `{ Bogdan: 75, Mia: 120 }`.

Acesta este un mod elegant și fără efecte secundare externe (exceptând mutarea internă a `acc` care e scopul reduce-ului). Observați că mutatez `acc` adăugând proprietăți; asta e normal, oricum `acc` e predat din nearly in reduce – ar fi putut teoretic să creez un nou obiect la fiecare pas cu rest operator, dar ar fi ineficient. Multe implementări de reduce pe obiect folosesc mutate acc oricum, considerându-l container local. E acceptabil în acest context.

10.6. **Var. F** — *keysfilterF.js* (`Object.entries` + `filter` + `Object.fromEntries`)

```
function filterObject(obj) {
  if (obj === null || typeof obj !== "object") {
    throw new TypeError("Argumentul trebuie să fie un obiect.");
  }
```

```
  const filteredEntries = Object.entries(obj)
    .filter(([key, value]) => typeof value === "number" && value > 50);
  return Object.fromEntries(filteredEntries);
}

console.log(filterObject({ Ana: 20, Bogdan: 75, Ion: 50, Mia: 120 }));
// -> { Bogdan: 75, Mia: 120 }
```

Explicație: Aceasta variantă combină abordarea D (filtrarea entries) cu reconstruirea unui obiect la final folosind **`Object.fromEntries(entriesArray)`** (metodă introdusă în ES2019). Practic, obținem `filteredEntries` ca array de perechi `[key, value]` ce trec testul. Apoi `Object.fromEntries(filteredEntries)` transformă acel array înapoi într-un obiect, atribuid fiecare key cu value corespunzător[20]. Obținem exact sub-obiectul filtrat.

Această metodă e foarte clară și reduce riscul de greșeli la reasamblare manuală. E și concisă. Singura atenționare: suportul fromEntries necesită un runtime modern (majoritatea motoarelor actuale îl au). In Node e disponibil de la v12 încolo. În 2025 cu siguranță e “safe” de folosit.

Rezultatul e același ca la A, B, E.

10.7. **Var. G** — *keysfilterG.js* (variantă CLI cu parametri)

```
"use strict";
function filterObject(obj, threshold) {
  return Object.fromEntries(
    Object.entries(obj).filter(([_, value]) => typeof value === "number" && value >
threshold)
  );
}

// Citire JSON și prag numeric din argumentele CLI
// Ex: node keysfilterG.js '{"Ana":20,"Bogdan":75,"Ion":50,"Mia":120}' 50
const [,, objArg, thresholdArg] = process.argv;
if (!objArg || !thresholdArg) {
  console.error("Utilizare: node keysfilterG.js <obiect_JSON> <prag_numeric>");
  process.exit(1);
}
let objInput;
try {
  objInput = JSON.parse(objArg);
} catch (e) {
  console.error("Obiectul furnizat nu este un JSON valid.");
  process.exit(1);
}
const threshold = Number(thresholdArg);
if (!Number.isFinite(threshold)) {
  console.error("Pragul trebuie să fie un număr.");
  process.exit(1);
}
console.log(filterObject(objInput, threshold));
```

Explicație: În această ultimă variantă, creăm un script CLI care primește două argumente: un obiect scris sub formă de literal JSON și un număr threshold. Scriptul parsează obiectul, parsează pragul numeric, apoi apelează `filterObject(obj, threshold)` pentru a obține obiectul filtrat pe criteriul "valoare > threshold".

Funcția `filterObject` folosește tehnica elegantă din var. F:

```
Object.entries(obj).filter(...).fromEntries(...).
```

Am generalizat predicatul să folosească `threshold` primit ca parametru (in loc de fix 50). Folosim _ ca nume de parametru pentru cheia pe care nu o folosim în condiție, ca convenție de "ignored".

Pentru input-ul dat în exemplu (fără console, modul de utilizare e dat ca comentariu), programul va afișa `{"Bogdan":75,"Mia":120}` (posibil fără spații, JSON stringify simplu). De fapt, cum folosim console.log pe un obiect, Node ar putea afișa-l într-un stil util, ex: `{ Bogdan: 75, Mia: 120 }` similar cu înainte, nu neapărat JSON-ul minimizat. Dacă am vrea JSON strict la output, am putea face `console.log(JSON.stringify(filterObject(objInput, threshold)))`. Dar nu e solicitat.

Implementarea CLI pe pasi:

- Citește `objArg` și `thresholdArg` din `process.argv`.
- Dacă lipsesc, cere utilizare corectă și iese cu cod 1.
- Tentează un `JSON.parse` pe objArg. Dacă eșuează (ex. nu e JSON valid), anunță eroare și iese.
- Convertește thresholdArg la număr, verifică că e finit numeric. Altfel eroare.
- Apoi apelează filterObject și afișează rezultat.

Am marcat codul cu `"use strict";`. Ca atare, dacă utilizatorul ar pune un obiect invalid (ex: neinchis cu { }, JSON.parse ar eșua), dacă pune un prag nepotrivit, etc., tratăm direct eroarea.

Observații: Această formă de input-output e puțin dificilă în CLI pentru obiecte complexe, de aceea am folosit JSON-ul. Ca alternativă, puteam citi dintr-un fișier JSON sau altceva, dar am respectat contextul folosind argument. E instructiv și cu JSON.parse.

În final, varietatea de implementări arată flexibilitatea JavaScript în manipularea structurilor de date. Varianta G este, în esență, o generalizare a variantei F într-un context real, modul aplicativ.

RECAPITULARE

În această secțiune am explorat numeroase aspecte legate de **funcții și array-uri în JavaScript** și aplicarea lor practică în diferite probleme. Să recapitulăm principalele idei și tehnici abordate:

- **FUNCȚII VARIADICE:** am discutat diferențele între abordările pre-ES6 (obiectul `arguments`) și cele moderne (parametrul *rest* `...args`), evidențiind avantajele clarității și tipizării array-urilor reale oferite de rest parameters[2]. Am trecut prin exemplul funcției `addToArray` în 6 variante (A-F), unde am văzut utilizarea `arguments`, a lui `...args` cu buclă vs. cu `push(...args)` vs. funcțional cu `[...array, ...args]` sau `concat`. Tabelul comparativ al acestor variante subliniază compromisurile între paradigmele imperativ/funcțional și mutație/imutabilitate (ex.: variantele E și F erau imutabile, returnând array nou, cu cost O(n+k) timp și memorie[21][22]).

- **ARRAY-URI ȘI METODE ESENȚIALE:** am folosit pe larg metode precum `Array.filter`[9], `Array.map`[16], `Array.reduce`. Aceste metode permit un stil declarativ, în care specificăm *ce* transformare sau criteriu aplicăm, iar biblioteca se ocupă de iterare. De exemplu, `filter` creează un sub-array cu elementele ce trec testul dat și nu modifică originalul[9], `map` creează un array nou cu rezultatele aplicării unei funcții asupra fiecărui element original[16]. Le-am aplicat în probleme ca *WordsFilter*, *MapMul* etc., comparându-le cu buclele tradiționale.

- **PARADIGMA IMPERATIVĂ VS. FUNCȚIONALĂ:** am evidențiat de mai multe ori aceste două stiluri. În mod imperativ, soluțiile noastre foloseau bucle (`for`, `while`, `for...of`, `for...in`) și acumulau rezultatul modificând variabile locale (ex.: construind un string caracter cu caracter, adăugând elemente într-un array rezultat). În mod funcțional, am preferat să folosim

funcții pure (fără efecte secundare externe), compunând transformări (ex.: `array.filter(...).map(...)`). Am subliniat că metodele declarative *nu modifică* array-ul original în cazurile noastre (ci creează copii), deci se pretează unui stil imutabil. Am discutat și costurile: de exemplu, crearea de array-uri noi are cost liniar în dimensiunea lor, deci de două ori mai multă memorie temporar în lanțuri `map(...).filter(...)` decât dacă am fi mutat in-place. Totuși, pentru lizibilitate și siguranța împotriva efectelor, adesea se preferă soluții funcționale dacă performanța nu e critică.

- **METODE MODERNE ALE OBIECTELOR GLOBALE:** am folosit `Object.keys`, `Object.entries` și `Object.fromEntries`. Acestea fac manipularea obiectelor mai ușoară:

  - `Object.keys(obj)` -> array cu cheile (propriile) ale obiectului[18];
  - `Object.entries(obj)` -> array de tupluri [cheie, valoare][19];
  - `Object.fromEntries(arrayOfEntries)` -> recreează un obiect dintr-o listă de perechi[23].

  Am văzut cum combinarea lor cu metode de array ne permite să filtrăm obiecte sau să extragem ușor anumite informații. De exemplu, `Object.entries(obj).filter(...).map(...)` e un pattern foarte puternic. În varianta Keys&Filters G, am folosit `Object.fromEntries(Object.entries.filter(...))` pentru a filtra direct un obiect pe baza valorilor sale, totul într-o manieră expresivă.

- **CLI și INTEGRARE Node.js:** la variantele notate cu H sau G în exemplele noastre, am exersat preluarea de argumente din linia de comandă cu `process.argv`, convertirea lor (de exemplu JSON.parse pentru obiecte, Number() pentru numere) și tratarea erorilor de input. Am arătat cum se poate construi interactiv un text (ex. un acrostih) sau filtra date (ex. obiect JSON) primite de la utilizator. Aceste exemple au scop didactic, dar ilustrează modul standard de a lucra cu argumente CLI în Node. Am evidențiat importanța validării intrării și a furnizării de mesaje de utilizare atunci când parametrii lipsesc sau sunt invalizi.

- **TRATAREA ȘIRURILOR ȘI A CARACTERELOR:** în probleme precum Acrostih sau WordsFilter, am folosit accesul la caractere prin `string[index]` și am menționat subtilități legate de Unicode. Deși în exemplele noastre nu am intrat adânc în Unicode, am adus aminte că `for...of` iterează corect caracterele (code points) chiar și compuse[5], pe când accesul cu index poate lua unități de cod individuale. Pentru simplificare, am presupus texte ASCII/latine (ceea ce e valabil pentru majoritatea stringurilor noastre exemplu). De asemenea, am folosit metode ca `String.split` și `String.padStart`, explicând utilitatea lor: `split` pentru tokenizare, `padStart` pentru formatarea numerelor cu zerouri (echivalent modern al adăugării de padding la stânga)[15].

- **RECURSIVITATE:** am inclus soluții recursive acolo unde se preta (Acrostih G, Formatter B, etc.) pentru a reaminti că orice algoritm repetitiv poate fi exprimat și recursiv. În JavaScript, recursivitatea excesivă poate duce la probleme de stivă, deci folosită cu prudență (nu am avut cazuri de recursivitate pe colecții extrem de mari). Totuși, ca instrument didactic, recursivitatea ajută la înțelegerea unor probleme (exemplu, acrostihul poate fi văzut ca litera din primul cuvânt + acrostihul restului de cuvinte).

- **PERFORMANȚĂ big-O:** am notat complexitățile aproximative la fiecare soluție. Majoritatea operațiilor de iterare complete a array-urilor sau proprietăților de obiect sunt O(n) față de mărimea colecției. Operații de concatenare repetată de string pot duce la costuri amortizate de O(n^2) dacă nu sunt optimizate intern (dar motoarele moderne optimizează destul de bine cazurile tipice de concatenare incrementală de stringuri scurte). În general, soluțiile noastre sunt eficiente pentru input-urile vizate (n fiind rezonabil – ex. număr de cuvinte, număr de elemente numeric). În probleme reale, e important de știut că metodele declarative (`filter`,

`map`) nu sunt magic mai eficiente decât buclele – ele tot iterează, deci complexitatea rămâne, însă aduc claritate și posibilitatea ca motorul JS să facă micro-optimizări.

- **BIBLIOTECI STANDARD vs. SOLUȚII CUSTOM:** am optat să folosim ce oferă limbajul (metode standard) în loc să reinventăm. De exemplu, la formatat numere am folosit `padStart` în varianta C în loc să scriem mereu bucle – e preferabil, fiind clar ce se întâmplă și delegând implementarea optimizată a acelui comportament către motorul JS. De asemenea, la combinarea de arrays (ex. push vs. spread vs. concat) am arătat opțiuni, accentuând că acolo unde semanticile coincid, preferința se duce spre codul mai clar sau idiomatic.

Încheind recapitularea: cunoștințele despre **funcții** (incluzând arrow functions, parametri rest, obiectul arguments), **structuri iterative** (bucle, HOF ca map/filter, recursivitate) și **manipularea colecțiilor** (array-uri și obiecte) sunt esențiale pentru a scrie cod JavaScript robust și lizibil. Am demonstrat că pentru același task pot exista multiple soluții, fiecare cu avantaje pedagogice sau practice. Familiarizarea cu aceste abordări îi va permite programatorului să aleagă soluția optimă în contextul dat – fie că optim înseamnă cea mai ușor de întreținut, fie cea mai performantă, fie un echilibru între acestea.

CHECKLIST (optional si ... individual) RECOMANDAT

- [x] **Funcții variadice:** Știu cum să definesc o funcție ce acceptă un număr variabil de argumente folosind sintaxa `...numeParam` (rest parameter) și cunosc diferența față de obiectul pre-ES6 `arguments`[2]. Am exersat implementări variadice (ex. adăugare elemente într-un array) atât imperativ, cât și funcțional.
- [x] **Operatorii *spread* și *rest*:** Înțeleg că `...` în context de apel extinde elementele unui array (spread)[4], iar în context de definiție de funcție colectează argumente într-un array (rest). Pot folosi spread pentru a concatena array-uri sau a copia obiecte în literali, și rest pentru a prelua restul argumentelor funcției.
- [x] **Metodele Array.prototype esențiale:** Stăpânesc folosirea lui `filter`, `map`, `reduce`, `forEach`, cunoscând ce fac și ce returnează fiecare. Știu că `filter` și `map` **nu modifică** array-ul original, ci creează unul nou[9], respectiv populat cu valori transformate[16]. Pot decide când să folosesc `reduce` (când am nevoie să agreg într-un rezultat de alt tip sau când vreau flexibilitate mare).
- [x] **Iterare modernă:** Pot itera direct elementele unui array cu `for...of` și cheile unui obiect cu `for...in`. Știu să folosesc `Object.keys(obj)` pentru a obține array-ul cheilor și `Object.entries(obj)` pentru perechi cheie-valoare, facilitând iterarea cu `for...of` pe aceste rezultate sau aplicații de metode array (`filter`, `map`).
- [x] **Manipularea obiectelor pe baza cheilor/valorilor:** Pot extrage cheile unui obiect cu anumite proprietăți ale valorilor sale (ex. cheile tuturor valorilor numerice peste X) folosind combinații ca `Object.entries(obj).filter(...).map(...)` sau pot obține direct un sub-obiect filtrat folosind `Object.fromEntries` pe rezultatul filtrat. Am verificat prin exemplu că știu filtra un obiect de scoruri.
- [x] **Formatarea și prelucrarea stringurilor:** Știu să folosesc `String.split` cu separator (sau regex) pentru a obține cuvinte dintr-o frază. Știu să obțin caracterul de la o anumită poziție cu `str[index]` sau `str.charAt(index)`. Pot construi un string incremental concatenând caractere sau folosind metode ca `Array.join` (ex. pentru acrostih). Știu de existența metodelor de *padding* (`padStart`, `padEnd`) introduse în ES2017 și le pot folosi pentru a formata numere cu zerouri la stânga[15].
- [x] **Recursivitate:** Pot scrie o soluție recursivă pentru o problemă de procesare a array-urilor (ex. calcul acrostih, adăugare de prefix de zerouri) identificând condiția de oprire și pasul de

reducere a problemei. Înțeleg limitările recursivității în JS (risc de depășire a stivei dacă lista e foarte lungă, lipsa TCO).

- [x] **CLI cu Node.js:** Știu să accesez argumentele din linia de comandă în scriptul Node folosind `process.argv`. Înțeleg că `process.argv[2]` corespunde primului argument real după numele scriptului. Pot converti argumentele la tipurile așteptate (folosind Number(), JSON.parse etc.) și pot implementa verificări/mesaje de eroare pentru input invalid. De asemenea, știu să folosesc `console.log` pentru a afișa rezultatele în consolă.
- [x] **Imprimarea și testarea soluțiilor:** Am testat soluțiile furnizate cu exemple cunoscute și am verificat că output-ul este cel așteptat. De exemplu, pentru *WordsFilter* am testat cu un șir de cuvinte și am verificat că variantele imperative și funcționale returnează același array filtrat; pentru *SquareDim* am calculat manual aria/perimetrul pentru o valoare și am verificat că toate variantele dau rezultat corect; pentru *Formatter* am testat cazuri unde numărul are mai puține cifre, exact atâtea sau mai multe decât width; pentru *Acrostih* am încercat listă de cuvinte normale și cu eventual elemente goale; pentru *Keys&Filters* am comparat vizual obiectul inițial și cel filtrat sau lista de chei rezultată.

Dacă toate elementele de mai sus sunt bifate cu înțelegere, atunci stăpânesc destul de bine subiectul funcțiilor și array-urilor în JavaScript, inclusiv metode și trucuri asociate.

ÎNTREBĂRI DE AUTO-EVALUARE (Quiz)

1. **Întrebare:** Care este diferența principală între obiectul special `arguments` și parametrii *rest* în JavaScript, și de ce este în general de preferat să folosim `...args` în locul `arguments`?

   Răspuns: Obiectul `arguments` este disponibil implicit în funcțiile tradiționale și conține toți parametrii transmiși, însă este *array-like* (nu are metodele unui Array) și devine neutilizabil în funcțiile *arrow*. Parametrul *rest* `...args` colectează argumentele suplimentare într-un **Array real**, asupra căruia putem aplica direct metode precum `map`, `filter`[2]. De asemenea, `arguments` are anumite particularități (în non-strict mode se sincroniza cu parametrii nominali, are `arguments.callee` etc. - caracteristici depășite), pe când *rest* e mai curat și mai sigur. Prin urmare, `...args` este preferat deoarece oferă un array normal, funcționează în strict mode și în funcții arrow, făcând codul mai modern și lizibil[3].

   Întrebare: Dacă `array1` este un array inițial, ce face următoarea linie de cod?

   ```
   let result = array1.filter(item => typeof item === "number").map(x => x * x);
   ```

   Răspuns: Aceasta linie compune două operații: mai întâi, `filter(item => typeof item === "number")` creează un nou array ce conține doar acele elemente din `array1` care sunt de tip număr (filtrează, deci elimină orice element ne-numeric). Apoi, pe rezultatul filtrat, se aplică `map(x => x * x)`, care produce un alt array nou obținând pătratul fiecărui element numeric selectat. Deci, în final, `result` va conține **pătratele numerelor** din `array1` (ignorând orice alte tipuri de elemente). De exemplu, dacă `array1 = ["cat", 2, 3, true]`, după execuție `result` va fi `[4, 9]`[24].

2. **Întrebare:** Adevărat sau fals – Metoda `Array.forEach` *returnează* un nou array constând din rezultatele aplicației funcției callback pe fiecare element.

Răspuns: *Fals*. `forEach` **nu returnează nimic (undefined)**; scopul său este de a executa o acțiune pentru fiecare element din array, fără a crea un array nou[25]. Metodele care creează și returnează array-uri noi sunt `map`, `filter`, `slice`, etc. Dacă vrem să obținem un array de rezultate, trebuie să folosim `map` în loc de `forEach`.

3. **Întrebare:** În codul următor, ce se afișează și de ce?

```
let x = 10;
function test(x, ...args) {
  console.log(args.length);
}
test(5, 6, 7);
```

Răspuns: La apelul `test(5, 6, 7)`, în interiorul funcției parametrul `x` va avea valoarea 5 (primul argument), iar `...args` va colecta restul argumentelor într-un array. `args.length` va reprezenta numărul de argumente *rest*. În apel avem 3 argumente totale, unul este alocat la `x`, deci rămân 2 pentru `args`. Așadar `console.log(args.length)` va afișa **2**. (Notă: variabila globală `let x = 10` nu influențează nimic aici, deoarece `x` din parametru maschează pe cea externă.)

4. **Întrebare:** Avem stringul `s = "Ăstazi"`. Ce probleme pot apărea dacă încercăm să iterăm caracterele cu un `for` clasic folosind indexare (de la 0 la `s.length-1`), comparativ cu un `for...of`?

Răspuns: Caracterul `"Ă"` (A cu accent circumflex) din `"Ăstazi"` este un caracter Unicode care în UTF-16 e reprezentat pe 1 cod unit (face parte din Latin Extended). În acest caz particular, atât `s[0]` cât și iterația `for...of` vor produce corect `"Ă"`. Însă, în general, pentru caractere multibyte sau compuse (emoji, unele simboluri), `for...of` va itera **fiecare caracter complet (punct de cod)**[5], în timp ce `for` cu indexe va itera cod unitățile. Astfel, cu `for` clasic riscăm să "desfacem" un caracter compus în două iterații diferite sau să obținem valori incoerente. `for...of` este de preferat pentru stringuri Unicode, asigurând parcurgerea corectă a caracterelor întregi. Deci, problema potențială e că indexarea poate trata incorect caracterele speciale, pe când `for...of` nu.

5. **Întrebare:** Ce face funcția `JSON.parse` și în ce context am folosit-o în exemplele de mai sus?

Răspuns: `JSON.parse` ia un șir de caractere (string) în format JSON și îl convertește (parsează) într-o valoare JavaScript corespunzătoare (obiect, array, număr, etc.). Practic, transformă textul JSON într-o structură de date utilizabilă în cod. În exemple, am folosit-o în varianta Keys&Filters G pentru a converti argumentul din linia de comandă (care venea ca text JSON pentru un obiect) în adevăratul obiect JavaScript pe care să-l putem filtra. Astfel, dacă rulam cu argument `'{"Ana":20,"Bogdan":75}'`, `JSON.parse` transforma acest string în obiectul `{ Ana: 20, Bogdan: 75 }` pe care apoi l-am prelucrat în script.

6. **Întrebare:** Cum putem obține rapid inversul operației lui `Object.entries`? Adică având un array de perechi `[cheie, valoare]`, să-l transformăm înapoi într-un obiect. Există vreo metodă standard?

   Răspuns: Da, metoda standard este **`Object.fromEntries(arrayOfEntries)`**[23]. Aceasta primește un array în care fiecare element este un sub-array de forma `[key, value]` (exact ca produsul lui `Object.entries`) și returnează un nou obiect ce are pentru fiecare astfel de pereche proprietatea `key` cu valoarea `value`. Am folosit `Object.fromEntries` în varianta F a Keys&Filters pentru a crea obiectul filtrat din array-ul de entries filtrate.

7. **Întrebare:** Avem următorul cod:

```
let arr = [1, 2, 3];
arr.reduce((acc, val) => acc + val, "");
```

   Ce rezultat produce și de ce?

   Răspuns: Aici folosim `reduce` cu valoare inițială `""` (un string gol) și callback-ul `(acc, val) => acc + val`. La prima iterație, `acc` este `""` și `val` este `1`, rezultând `"1"` (string). Apoi `acc` devine `"1"`, `val` următor `2`, deci `"1" + 2 -> "12"` (concatenare, 2 e convertit la string implicit). Ultima iterație: `acc = "12"`, `val = 3`, obținem `"123"`. Deci rezultatul final al expresiei este stringul **"123"**. Așadar, reduce a *concatenat* elementele array-ului într-un șir, deoarece inițializarea a fost un string. (Acest exemplu arată și conversia automată la string când se adună string + număr.)

8. **Întrebare:** În ce situație folosirea unei bucle `for` clasice ar putea fi preferată în locul metodei `forEach` sau a altor metode de iterare moderne? Dă un exemplu.

   Răspuns: Bucle `for` clasice pot fi preferate în situații unde avem nevoie să controlăm strict indicele de iterare, să modificăm fluxul (ex: să facem `break` sau `continue` ușor) ori pentru ușoare avantaje de performanță la colecții foarte mari. De exemplu, dacă vrem să parcurgem un array până găsim un anumit element și apoi să ne oprim imediat, un `for` clasic cu `break` este foarte direct. Cu `forEach` nu putem folosi `break` (ar trebui subterfugii). Alt exemplu: dacă vrem să parcurgem doi array-uri în paralel prin index, `for` clasic e convenabil (`for(let i=0; i<n; i++) { ... arr1[i], arr2[i] ... }`). Deci, când avem nevoie de un control mai fin asupra iterării sau de evitare a costurilor suplimentare (evitarea creării de closure pentru callback la fiecare element, deși asta e minor), `for` poate fi preferat.

9. **Întrebare:** Citește următorul fragment și spune ce afișează și de ce:

```
const obj = { a: 1, b: 2 };
Object.defineProperty(obj, 'c', { value: 3, enumerable: false });
console.log(Object.keys(obj));
```

Răspuns: `Object.defineProperty` cu `enumerable: false` face ca proprietatea `'c'` să nu fie enumerabilă (nu apare la iterare normală). Astfel, `obj` are cheile proprii `'a'` și `'b'` ca enumerabile, și `'c'` ca neenumerabilă. `Object.keys(obj)` returnează **doar** cheile enumerate ale obiectului[18]. Prin urmare, `console.log(Object.keys(obj))` va afișa `["a", "b"]`. Cheia "c" nu apare în acest array deoarece este neenumerabilă.

BIBLIOGRAFIE

1. **MDN Web Docs – Functions – Rest parameters**. (2023, July 8). *Mozilla Developer Network*. Explică sintaxa și comportamentul parametrilor rest (`...args`), comparându-i cu obiectul `arguments`[26][27]. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters

2. **MDN Web Docs – Spread syntax (...)**. (2023). *Mozilla Developer Network*. Descrie operatorul spread și utilizările sale în liste de argumente, array literals și object literals, evidențiind diferența față de rest (spread "expandează", rest "condensează")[4]. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

3. **MDN Web Docs – Array.prototype.filter()**. (2022). *Mozilla Developer Network*. Documentația metodei `filter`, subliniind că returnează un nou array cu elementele ce trec testul, fără a modifica originalul[9]. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

4. **MDN Web Docs – Array.prototype.map()**. (2022). *Mozilla Developer Network*. Documentația metodei `map`, explicând că produce un array populat cu rezultatele funcției aplicate fiecărui element al array-ului original, pe care nu-l schimbă[16]. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

5. **MDN Web Docs – Object.keys()**. (2023). *Mozilla Developer Network*. Specifică faptul că `Object.keys` întoarce un array cu cheile proprii (enumerable) ale obiectului[18]. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys

6. **MDN Web Docs – Object.entries()**. (2023). *Mozilla Developer Network*. Precizează că `Object.entries` întoarce un array de perechi `[cheie, valoare]` pentru fiecare proprietate enumerabilă proprie a obiectului[19]. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries

7. **MDN Web Docs – Object.fromEntries()**. (2019). *Mozilla Developer Network*. Documentația metodei introduse în ES2019, care construiește un obiect dintr-un array de entități tip `[key, value]`[23]. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/fromEntries

8. **ECMAScript 2015 Specification – Array.prototype.forEach**. (2015). *ECMA-262 6th Edition*. Standardul definește că `forEach` invocă funcția furnizată pentru fiecare element din array în ordine și nu returnează un nou array[25]. URL: https://www.ecma-international.org/ecma-262/6.0/#sec-array.prototype.foreach

9. **MDN Web Docs – String.prototype.padStart()**. (2023). *Mozilla Developer Network*. Explică metoda `padStart`, care completează stringul la o lungime specificată prin adăugarea unui padding la început[15]. Include exemple de folosire pentru a adăuga zerouri conducătoare la

numere (similar cu `printf` format)[13]. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/padStart

10. **MDN Web Docs – for...of statement**. (2023). *Mozilla Developer Network*. Descrie instrucțiunea `for...of`, menționând că iterează peste *valorile* sursei (iterable) – în cazul stringurilor, peste caracterele lor Unicode complete[28][29]. Relevanța aici e pentru discuția despre iterația corectă a șirurilor cu diacritice. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of

[1] [10] Functions - JavaScript | MDN

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

[2] [3] [26] [27] Rest parameters - JavaScript | MDN

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

[4] Spread syntax (...) - JavaScript | MDN

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

[5] [6] [7] [8] ___seminar2video6_explicativ_6freq.docx

file://file_00000000dfe86243af294c9393ed022e

[9] [25] Array.prototype.filter() - JavaScript | MDN

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

[11] [12] [13] [14] [15] String.prototype.padStart() - JavaScript | MDN

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/padStart

[16] Array.prototype.map() - JavaScript | MDN

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

[17] [18] [19] NETACAD JavaScript Essentials 2.docx

file://file_000000000fb8620abfcae73dd79fb413

[20] [21] [22] ___seminar2video4_explicativ_4arguments.docx

file://file_000000006b646246b13924fbba1d38da

[23] Unicode character class escape: \p{...}, \P{...} - JavaScript | MDN

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Regular_expressions/Unicode_character_class_escape

[24] NETACAD JavaScript Essentials 2.docx

file://file_00000000ecc0620a9da963ae5f801db9

[28] for...of - JavaScript | MDN - Mozilla

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

[29] Difference between (for... in) and (for... of) statements?

https://stackoverflow.com/questions/29285897/difference-between-for-in-and-for-of-statements