De la bucle imperative la pipeline-uri declarative

- Seminar 3 – very short notes of mine -

INTRODUCERE & SCENARIU

Seminarul de astăzi este despre Funcții și Array-uri în JavaScript. În următoarele 45 de minute vom explora împreună cum putem scrie un cod mai curat, mai expresiv și mai modern.

Vom pune accent pe practicile moderne ES6+ și vom vedea cum putem prelucra colecții de date, cum ar fi array-urile, folosind două abordări fundamental diferite: cea imperativă și cea funcțională.[1]

Tema centrală de astăzi este trecerea de la a-i spune computerului *cum* să facă ceva, pas cu pas - ceea ce numim programare **imperativă** - la a-i spune *ce* rezultat dorim să obținem, lăsând detaliile de implementare în seama limbajului. Aceasta este esența programării **declarative** sau funcționale în JavaScript. Vom vedea cum această schimbare de perspectivă nu doar că ne scurtează codul, dar îl face mai ușor de citit, de testat și de întreținut.

Vă invit să deschideți editorul vostru de cod preferat, eu voi folosi Visual Studio Code, și să creați un fișier nou, de exemplu seminar3.js. Vom scrie și rula codul împreună.

PRIMUL PAS: de la bucle manuale la filtrare declarativă

Să începem cu o problemă concretă. Imaginați-vă că primim o frază ca un string și trebuie să extragem din ea toate cuvintele care au o lungime de cel puțin 3 caractere. De exemplu, din fraza 'ana are mere și prune', vrem să obținem un array care conține doar ['mere', 'prune'].[1]

Abordarea Imperativă - Bucla for

Să scriem împreună prima variantă, cea clasică, pe care probabil o cunoașteți deja.

```
function filterWordsLongerThan3_Imperative(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;
}

console.log(filterWordsLongerThan3_Imperative("ana are mere și prune"));
```

Să analizăm fiecare linie de cod.

- function filterWordsLongerThan3_Imperative(text): Definim o funcție clasică ce acceptă un parametru, text.

- const words = text.split(' ');: Primul pas este să spargem fraza într-un array de cuvinte. Metoda .split(' ') face exact asta, folosind spațiul ca separator.
- const result =;: Acum, **declarăm un array gol**, result. Acesta este containerul în care vom acumula manual rezultatele noastre. Acesta este un pas tipic în programarea imperativă: **creăm o stare pe care o vom modifica pe parcurs**.
- for (let i = 0; i < words.length; i++): Aici este nucleul logicii imperative: o buclă for clasică. Îi spunem explicit motorului JavaScript: 'începe cu un index i de la 0; continuă atâta timp cât i este mai mic decât lungimea array-ului; și incrementează i după fiecare pas'. Noi controlăm fiecare detaliu al iterației.
- if (words[i].length >= 3): În interiorul buclei, pentru fiecare cuvânt în parte, accesat prin words[i], verificăm condiția noastră: dacă lungimea lui este mai mare sau egală cu 3.
- result.push(words[i]);: Dacă condiția este adevărată, folosim metoda .push() pentru a adăuga cuvântul în array-ul nostru result. Observați cuvântul cheie aici: **mutație**. Modificăm direct, *in-place*, array-ul result.
- return result;: La final, după ce bucla a parcurs toate elementele, returnăm array-ul result completat.

Dacă rulăm acest cod, vom vedea în consolă exact ce ne așteptam: ['mere', 'prune']. Funcționează perfect, dar este verbos și ne obligă să gestionăm manual multe detalii.

Abordarea Declarativă - Array.prototype.filter

Acum, să rescriem funcția folosind o abordare modernă, funcțională.

```
const filterWordsLongerThan3_Declarative = (text) => {
  return text.split(' ').filter(word => word.length >= 3);
};

console.log(filterWordsLongerThan3_Declarative("ana are mere și prune"));
```

Să analizăm și această variantă.

- const filterWordsLongerThan3_Declarative = (text) =>...: Acum, folosim sintaxa modernă ES6. Definim o constantă și îi asignăm o funcție săgeată (arrow function). Este o alternativă mai concisă la cuvântul cheie function.
- return text.split(' ').filter(...): La fel ca înainte, începem cu .split(). Dar acum, în loc de o buclă for, înlănțuim direct metoda .filter().
- filter(word => word.length >= 3): Acesta este momentul magic. Metoda .filter() este o funcție de ordin înalt (Higher-Order Function) - adică o funcție care primește altă funcție ca argument.[1] Funcția pe care i-o dăm, word => word.length >= 3, se numește **predicat**. Pentru fiecare element din array (pe care îl numim word), acest predicat returnează true sau false. Metoda .filter() se ocupă intern de toată logica de iterare și construiește **un nou array** care conține doar elementele pentru care predicatul a returnat true.

Observați diferența fundamentală: nu am mai creat un array gol, nu am mai gestionat un index i. Am declarat doar ce vrem: 'din acest array de cuvinte, filtrează-le și păstrează-le doar pe cele a căror lungime este >= 3'. Rulând codul, obținem același rezultat, dar cu un efort cognitiv mult mai mic.

PERSPECTIVE & COMPARAȚII

Să analizăm ce s-a întâmplat. În prima variantă, am descris cum să se facă filtrarea, pas cu pas. În a doua, am descris ce rezultat vrem. Codul declarativ este mai aproape de limbajul uman și ne permite să ne concentrăm pe logica problemei, nu pe mecanica buclelor.

O altă diferență crucială este **imutabilitatea**. În exemplul cu .filter(), array-ul original de cuvinte nu este

niciodată modificat. Metoda .filter() returnează un array complet nou.[1] În varianta imperativă, am modificat activ array-ul resultat. De ce este important? Într-o aplicație mare, modificarea datelor în mod neașteptat (efecte secundare) poate duce la bug-uri foarte greu de depistat. Lucrând cu structuri de date imutabile, codul nostru devine mai previzibil și mai sigur.

Pentru a clarifica, iată o comparație directă:

Criteriu	Abordare Imperativă (for loop)	Abordare Declarativă (.filter)
Stil	**Cum** se face (algoritm pas-cu-pas)	**Ce** se dorește (rezultatul final)
Lizibilitate	Mai verbos, necesită urmărirea logicii buclei	Concis, intenția este clară la prima vedere
Managementul Stării	Mutație (modificarea array-ului result cu .push())	Imutabilitate (crearea unui array nou)
Risc de Erori	Mai mare (ex: erori de index, off-by-one)	Mai mic (logica de iterare este abstractizată)

TRANSFORMAREA DATELOR CU Array.prototype.map

Bun, am învățat să selectăm elemente. Dar ce facem dacă vrem să transformăm fiecare element dintr-un array? De exemplu, să luăm un array de numere și un factor de multiplicare, să zicem 5, și să obținem un nou array.[1]

Am putea, desigur, să folosim din nou o buclă *for*, să creăm un array gol și să facem push cu rezultatul înmulțirii la fiecare pas. Dar acum știm că există o cale mai bună.

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

console.log(multiplyArray(, 5));
console.log(multiplyArray([-2, 0, 7], 3));
```

Să disecăm această linie.

- const multiplyArray = (arr, factor) =>...: Din nou, folosim o funcție săgeată concisă pentru a defini funcția noastră.
- arr.map(x => x * factor): Aici intervine Array.prototype.map. La fel ca .filter, .map este o funcție de ordin înalt care iterează peste fiecare element. Diferența fundamentală este că funcția pe care i-o dăm ca argument nu trebuie să returneze true sau false. În schimb, ea trebuie să returneze **noua valoare** pentru elementul respectiv.
- Deci, pentru fiecare x din array, noi returnăm x * factor. Metoda .map colectează toate aceste noi valori și le pune într-un **array nou**, pe care îl returnează.[1]
- O regulă importantă: array-ul returnat de .map va avea **întotdeauna aceeași lungime** ca array-ul original. Este o transformare 1-la-1.

Rulând codul, vedem rezultatele: [5, 10, 15] și [-6, 0, 21]. Simplu, elegant și expresiv.

UNEALTA SUPREMĂ[1] PENTRU Array-uri: Agregarea cu reduce

Am văzut cum să selectăm (filter) și cum să transformăm (map). Dar ce se întâmplă când vrem să "fierbem" un întreg array pana scade la o singură valoare? De exemplu, să luăm un array de cuvinte precum ['Mere', 'Ana', 'Are'] și să construim un singur string din prima literă a fiecărui cuvânt, adică 'MAA'. Această operație se numește agregare sau reducere.[1]

```
function createAcrostic(words) {
  return words.reduce((accumulator, currentWord) => {
    if (typeof currentWord === "string" && currentWord.length > 0) {
      return accumulator + currentWord;
    }
    return accumulator;
  }, "");
}

console.log(createAcrostic(["Mere", "Ana", "Are"]));
```

Array.prototype.reduce este probabil cea mai versatilă metodă de array. Poate face tot ce fac map și filter, și mult mai mult. Să o descompunem cu atenție.

Metoda reduce acceptă două argumente principale:

1. O funcție 'reducer' (callback).
2. O valoare inițială pentru acumulator.
   - }, "");: Să începem cu al doilea argument: "" (un string gol). Aceasta este **valoarea inițială**. Ea stabilește atât valoarea de pornire a procesului, cât și tipul de date al rezultatului final. Pentru că vrem să construim un string, pornim cu un string gol.
   - reduce((accumulator, currentWord) => {... }): Acum, funcția noastră 'reducer'. Ea primește cel puțin doi parametri: accumulator și currentWord.
   - accumulator: Este valoarea acumulată până în acest punct. La prima iterație, accumulator va fi egal cu valoarea inițială pe care am furnizat-o, adică "".
   - currentWord: Este elementul curent din array care este procesat, similar cu word din filter sau x din map.

Să urmărim execuția pas cu pas:

1. **Iterația 1:** accumulator este "". currentWord este 'Mere'. Condiția este adevărată. Funcția returnează "" + 'M', adică 'M'. Această valoare returnată devine noul accumulator pentru următoarea iterație.
2. **Iterația 2:** accumulator este acum 'M'. currentWord este 'Ana'. Condiția este adevărată. Funcția returnează 'M' + 'A', adică 'MA'. Acesta este noul accumulator.
3. **Iterația 3:** accumulator este 'MA'. currentWord este 'Are'. Condiția este adevărată. Funcția returnează 'MA' + 'A', adică 'MAA'.
4. Array-ul s-a terminat. Valoarea finală a acumulatorului, 'MAA', este returnată de reduce.

Rulând codul, obținem 'MAA'. Am redus un array de stringuri la un singur string, într-un mod declarativ și elegant.

[1] “That Tool to rule them all”

Dincolo de Array-uri: Manipularea Obiectelor

Aceste metode sunt fantastice, dar ele există pe Array.prototype. Ce facem când avem un obiect și vrem să aplicăm o logică similară? De exemplu, avem un obiect cu scorurile jucătorilor și vrem să păstrăm doar jucătorii cu scorul peste 50.[1]

```
const scores = { Ana: 20, Bogdan: 75, Ion: 50, Mia: 120 };
// Rezultatul așteptat: { Bogdan: 75, Mia: 120 }
```

Secretul este să transformăm temporar obiectul într-un array, să folosim uneltele pe care tocmai le-am învățat, și apoi să transformăm array-ul înapoi într-un obiect. Vom crea o 'conductă' (pipeline) de procesare a datelor.

```
function filterScores(scoresObject) {
  // Pasul 1: Deconstruim obiectul într-un array
  const entries = Object.entries(scoresObject);
  console.log("Pasul 1 - După Object.entries:", entries);

  // Pasul 2: Filtrăm array-ul folosind metodele cunoscute
  const filteredEntries = entries.filter(([key, value]) => value > 50);
  console.log("Pasul 2 - După filter:", filteredEntries);

  // Pasul 3: Reconstruim obiectul din array-ul filtrat
  const resultObject = Object.fromEntries(filteredEntries);
  return resultObject;
}

console.log("Rezultat final:", filterScores(scores));
```

Să parcurgem pașii:

- **Pasul 1: Object.entries(scoresObject)**. Această metodă magică ia un obiect și îl transformă într-un array de array-uri, unde fiecare sub-array conține o pereche [cheie, valoare].[1] După cum vedeți în consolă, obținem ['Ana', 20],, ['Ion', 50], ['Mia', 120].
- **Pasul 2: .filter(([key, value]) => value > 50)**. Acum că avem un array, putem folosi .filter()! Observați sintaxa ([key, value]) în parametrul funcției săgeată. Aceasta se numește **destructurare** și ne permite să dăm nume direct elementelor din sub-array-uri, făcând codul foarte lizibil. Filtrăm pe baza value > 50. Rezultatul intermediar este , ['Mia', 120].
- **Pasul 3: Object.fromEntries(filteredEntries)**. Aceasta este operația inversă a lui entries. Ia un array de perechi [cheie, valoare] și construiește un obiect nou din ele.[1]

Acum, să punem totul la un loc într-o singură linie, pentru a vedea cât de expresiv poate fi acest lanț:

```
const filterScoresConcise = (scoresObj) =>
  Object.fromEntries(
    Object.entries(scoresObj).filter(([key, value]) => value > 50)
  );

console.log("Rezultat concis:", filterScoresConcise(scores));
```

Am creat un pipeline de date puternic și lizibil, transformând un obiect într-un array pentru procesare și apoi înapoi într-un obiect pentru rezultat.

CONCLUZII

Astăzi am explorat trei piloni ai programării funcționale în JavaScript:

- **.filter()**: Pentru a **selecta** elemente. Creează un sub-set al array-ului original. Rezultatul poate fi mai scurt.
- **.map()**: Pentru a **transforma** elemente. Creează un nou array cu aceeași lungime, dar cu elemente modificate.
- **.reduce()**: Pentru a **agrega** elemente. 'Fierbe' un array la o singură valoare (care poate fi un număr, un string, un array, sau chiar un obiect).

Am văzut cum aceste unelte ne ajută să trecem de la un stil **imperativ**, în care descriem fiecare pas, la un stil **declarativ**, în care ne exprimăm intenția. Acest lucru duce la un cod mai scurt, mai lizibil și, cel mai important, mai previzibil, datorită principiului **imutabilității**.

Vă încurajez să luați exemplele din materialul de curs și să încercați să le rezolvați folosind aceste trei metode. Practica este esențială pentru a vă simți confortabil cu acest stil de programare.

Vă mulțumesc pentru atenție. Acum: întrebări?

Lucrări citate

1. S2verB (nextlab but simplier) Funcții și Array-uri în JavaScript.pdf