**Seminarul 2 — Partea 1 (Teorie, extins)**

Seminarul 2 — Partea 1 (Teorie, extins)

Titlu: De la liste brute la informație utilă — funcții și array‑uri în JavaScript (map / filter / reduce, utilitare, chaining)

Hook realist (situație de lucru)

Imaginați‑vă că sunteți coordonatorul unei asociații studențești dintr‑o facultate mare. În fiecare săptămână primiți exporturi CSV și JSON dintr‑un formular de înscriere pentru evenimente: hackathoane, workshop‑uri, seri de mentorat. Coloanele sunt aparent simple: name, email, faculty, year, interests, registered\_at, preferințe de orar. Dar în realitate apar tot felul de variații: facultatea e scrisă „CSIE”, „csie” sau „C.S.I.E.”; interesele vin uneori ca „AI|Web|UX”, alteori ca o listă JSON ["AI", "Web", "UX"]; anii sunt 1, „I”, „1.0” sau chiar stringuri goale; emailurile conțin spații la capete; uneori apar duplicate la aceeași persoană care a completat din greșeală formularul de două ori. Deși panoul grafic al platformei are câteva rapoarte, întrebare după întrebare te trimite spre exportul brut: „Care sunt top‑5 interese reale pe ultima lună, dar excluzând duplicatele și înscrierile test?”, „Câte intrări valide avem pe fiecare facultate și an? Poți să ne dai un CSV sumar?”, „Ce orar preferă majoritatea pentru a programa un atelier pilot?”

Într‑un astfel de context, ceea ce separă o prelucrare fragilă de una robustă este disciplina cu care tratezi listele de obiecte. Practic, vei construi un „bandă de asamblare” declarativă din operații elementare pe array‑uri: map (transformă fiecare element), filter (păstrează doar elementele acceptate de predicat), reduce (agregă în structuri sumar). De aici, îți proiectezi treptat o mică bibliotecă de utilitare (groupBy, countBy, keyBy, partition, uniqBy, difference, intersection, chunk, zip, flatten) și înveți să le compui („chaining”) astfel încât fiecare pas să fie lizibil, testabil și refolosibil. Rezultatul este repetabil: dacă peste două luni fixezi alte criterii, doar schimbi parametrii sau compoziția de pași, nu rescrii totul de la zero.

1. De ce gândire funcțională pe colecții? (motivație conceptuală)

Gândirea funcțională se sprijină pe trei idei operaționale utile pentru prelucrări de date:

• Funcții pure. O funcție pură este deterministă (pentru aceleași intrări dă mereu aceleași ieșiri) și nu are efecte secundare observabile. Aceasta aduce predictibilitate și ușurează testarea unit (unit testing). Dacă o funcție care calculează frecvențe modifică în același timp o variabilă globală „hidden”, testele devin fragile. În schimb, o funcție pure returnează pur și simplu un obiect rezultat (de ex., un dicționar de frecvențe) fără a altera mediul din jur.

• Compoziție. Funcțiile se compun: „ieși din A pentru a intra în B”. În loc să scrii o funcție enormă care face „de toate”, îți definești pași mici cu un scop clar: normalizeazăIntrare → filtreazăValide → proiecteazăCâmpuri → agregăFrecvențe → sortează → taieTopK. Fiecare pas poate fi testat și înlocuit fără a perturba celelalte, ceea ce duce la mentenanță ieftină.

• Transparență referențială. O expresie poate fi înlocuită cu rezultatul ei fără a schimba comportamentul programului. Această proprietate simplifică raționamentele și face codul „portabil mental”: îl poți muta, împărți, reorganiza fără „surprize” legate de stări ascunse.

În operații pe array‑uri, aceste idei se traduc într‑un „alfabet” de transformări. Un avantaj major pe care îl vei simți practic este că transformările compuse (map/filter/reduce) fac vizibile ipotezele tale: unde normalizezi, unde excluzi date, unde agregi. Când un coleg te întreabă „de ce iese top‑K = {AI, Web, Data, UX, Finance}?”, îi poți indica, pas cu pas, invarianta și transformările intermediare; nu trebuie să „reconstruiești” mental un amestec de bucle imbricate.

Analogie cotidiană — liniile de sortare: imaginează o bandă pe care vin cutii (intrările tale). La prima stație (map) lipsești o etichetă clară (convertești „csie” în „CSIE”, spargi „AI|Web” în [„AI”, „Web”], tai spațiile). La stația de control (filter) lași doar cutiile ce trec testul minim (facultate validă, interes nenul). La următoarea stație, un operator numără câte cutii sunt pe fiecare traseu (reduce către frecvențe). Șeful de tură alege apoi primele K rezultate pe baza unei reguli clare (sort + slice). Dacă trebuie adăugată o nouă stație (de ex., deduplicare), nu închizi fabrica: inseri o stație în lanț.

2. Recapitulare strict necesară (JavaScript modern) — „terenul” pe care jucăm

2.1 Variabile, domenii, TDZ

• Preferă const pentru legături imutabile, let pentru reatribuiri controlate; var este evitat în codul modern.

• „Temporal Dead Zone” (TDZ) interzice accesul la let/const înainte de inițializare. În pipelines, evitați artificii care depind de hoisting neclar — exprimați dependențele explicit.

• Pentru colecții, evitați mutațiile „in‑place” (e.g., sort fără clonă) când scrii funcții „de bibliotecă”. Mutația e permisă la margini (I/O), dar corpul logic ar trebui să lucreze cu copii; sau specificați clar în contract dacă funcția modifică argumentele (și testați acest comportament).

2.2 Funcții — arrow vs. function

• Arrow functions (=>) au this lexical (nu‑și „propriu‑l” this), iar concizia lor le face ideale pentru map/filter/reduce. function clasic rămâne util pentru denumiri și pentru situații în care hoistingul controlat și contextul de apel au relevanță. În biblioteci de utilitare, preferința uzuală: arrow pentru transformări scurte, function denumită pentru agregări non‑triviale.

• Parametri cu valori implicite, rest (...) și destructurarea în semnături cresc lizibilitatea: semnătura devine o „mini‑spec” a contractului.

2.3 Tipuri și conversii explicite

• Din fișiere CSV/JSON reale, numericele vin adesea ca stringuri; booleanii ca „true”/„false”; datele în formate heterogene („2025‑09‑01”, „01/09/2025”). Normalizați explicit (Number, parseInt cu bază, Boolean, Date.parse sau biblioteci dedicate). Evitați să vă bazați pe coerciuni implicite (de ex., "10" < "2" este true lexical, dar numeric nu).

• Tratați lipsa de valori (null/undefined) ca un caz normal, nu excepțional: scrieți predicatul isDefined și folosiți‑l în filter/compact.

2.4 Array‑uri și metodele lor

• Array este secvența fundamentală în JS. Metodele sale definitorii: map, filter, reduce, some, every, find, sort, slice, flat, flatMap. Pentru transformări predictibile, considerați‑le „cărămizile” de bază.

• Evitați efecte secundare în callback‑urile acestor metode. Dacă aveți nevoie de logare (diagnostic), intercalează un „tap” (vezi mai jos).

3. map / filter / reduce — semnături, contracte, legi de rescriere

3.1 map (transformare element cu element)

Semnătură: map(f: (T) → U): T[] → U[]

Contracte:

• Dimensiunea: |map(f)(xs)| = |xs| (același număr de elemente).

• Ordinea: ordinea inițială este păstrată.

• Erori: dacă f aruncă, map propagă; folosește try/catch în f doar dacă vrei comportament „rezilient” (de ex., înlocuiește cu o valoare sentinelă).

Cazuri‑limită:

• Lista goală: map(f)([]) = [].

• f identitate: map(x ⇒ x) = identitate.

• Efecte secundare în f: nedorite în general (testele devin dependențe de ordinea apelurilor).

3.2 filter (selectare prin predicat)

Semnătură: filter(p: (T) → boolean): T[] → T[]

Contracte:

• Submulțime ordonată a listei inițiale (sub‑secvență).

• Predicatul p nu trebuie să aibă efecte; altfel lanțul devine impredictibil.

Cazuri‑limită:

• Lista goală produce listă goală (invariantă).

• p ≡ true → identitate; p ≡ false → listă goală.

• Atenție la predicat cu coerciuni „surpriză” (ex.: !!"0" e true).

3.3 reduce (agregare)

Semnătură: reduce(reducer: (Acc, T) → Acc, init: Acc): T[] → Acc

Contracte:

• Parcurge secvența și acumulează într‑un Acc (poate fi număr, obiect, hartă, tuplu).

• Inițializarea contează: reduce fără init pe listă goală este eroare; cu init, definești explicit cazul de bază.

Cazuri‑limită:

• O singură trecere. Poți agrega mai multe valori simultan (ex.: total, min, max, count) într‑un singur obiect acumulat — adesea mai eficient decât treceri multiple.

• Asociație și identitate: dacă reducerul este asociativ și există un element neutru, ai deschis drumul paralelizării și reordonării (idee exploatată în sisteme la scară).

3.4 Legi de rescriere („algebra” practică a listelor)

• Fusioane: map(f) ∘ map(g) ≡ map(f ∘ g). Reducere de treceri fără a pierde claritatea (atenție: debug poate fi mai dificil; folosiți „tap”).

• Filtre: filter(p) ∘ filter(q) ≡ filter(x ⇒ p(x) && q(x)).

• Distribuții: anumite map/filter pot fi rearanjate fără a schimba rezultatul (dacă p nu depinde de transformarea lui map). Construiți o intuiție și validați prin teste.

Notă pragmatică: codul didactic pune accent pe lizibilitate. Folosiți legile de rescriere pentru optimizare numai după ce aveți testele verzi și măsurători care indică un beneficiu clar.

4. Utilitare pe colecții — o trusă de bază (contracte, complexități, capcane)

4.1 groupBy(keyFn)

• Produce Record<string, T[]> (sau Map<K, T[]>). Complexitate O(n); memorie O(n).

• Chei compuse: fie serializați tuplurile (faculty|year), fie folosiți Map cu chei obiect (atenție la egalitate de referință).

• Capcană: distribuții dezechilibrate (o cheie uriașă) pot crea liste foarte mari într‑un singur „bucket”; luați în calcul skippere sau sortare prealabilă dacă doriți paginare per cheie.

4.2 countBy(keyFn)

• Produce Record<string, number> (sau Map<K, number>) cu frecvențe; O(n).

• Pattern clasic: pentru interese multiple per rând, folosiți flatMap înainte de countBy.

• Notă: convertiți rezultatul în listă de perechi (entries) pentru sortare ușoară.

4.3 keyBy(keyFn)

• Produce Record<string, T> (chei unice). Decideți politica la coliziuni: „first wins”, „last wins”, „throw”. Documentați decizia și testați‑o.

• Când datele pot avea duplicate reale, preferați groupBy în locul lui keyBy.

4.4 partition(p)

• Produce [acceptate, respinse]; util pentru contabilizarea „pierderilor” din pipeline (calitate de date).

4.5 uniq / uniqBy

• Elimină duplicatele; O(n) cu Set/Map. Decideți noțiunea de egalitate (case‑insensitive pe e‑mail? spații ignorate?). Orice decizie trebuie testată.

4.6 difference / intersection (pe mulțimi)

• difference(A,B): elemente din A care nu apar în B; intersection: elemente comune. Implementare eficientă: convertiți B într‑un Set.

• Când elementele sunt obiecte, extrageți o cheie canonică (id sau o proiecție serializată stabilă).

4.7 chunk(n)

• Împarte lista în bucăți de dimensiune n (ultimul chunk poate fi mai mic). Bun pentru paginare sau pentru a evita limite în API‑uri (e.g., „max 100 item‑uri per cerere”).

4.8 zip / unzip

• zip alătură listele pozițional; cere lungimi comparabile sau o politică la lungimi diferite (truncate vs. padding).

4.9 flatten / flat / flatMap

• Pentru liste de liste, flatMap este adesea mai elegant decât map + flat. Atenție la nivelul de „adâncime” al flatten (flat(1) vs. flat(Infinity)).

4.10 compact / defined

• Elimină null/undefined (și, opțional, NaN sau stringuri goale). Fixează explicit regulile, altfel „curățarea” poate ascunde date legitime.

4.11 sortare cu comparator compus

• Ordinea primară: după frecvență descendent; a doua: alfabetic (pentru determinism). Nu vă bazați pe „întâmplarea” motorului — implementați tie‑breakers explicite. Pentru text multilingv, luați în calcul Intl.Collator (cost mai mare, dar ordonare „umană”).

4.12 normalizare și canonicizare

• name/email: trim, lowercase (sau casefold), eliminare diacritice după caz; regex pentru validare.

• faculty: enumerare de valori acceptate; mapări „csie” → „CSIE”.

• interests: lista clară (string → listă; delimitatori de tip „|”, „;”, „,”).

5. Chaining & compoziție — proiectarea „rețetelor”

5.1 De ce chaining?

• Pentru că ordinea din cod reflectă ordinea transformărilor. În plus, fiecare etapă poate fi inspectată/„tap‑uită” ușor.

• Dezavantaj: reutilizarea parțială e mai dificilă; compensați cu funcții‑rețetă (pipe/compose).

5.2 pipe/compose și „tap/peek”

• pipe(f1, f2, f3) produce o funcție nouă. Avantaj: rețete „nume” (de ex., buildTopKReport) care pot fi așteptate în teste (contract clar).

• tap(fn) — execută fn(x) pentru debugging și returnează x neschimbat. Bun pentru a verifica invarianta după normalizare sau după filtrare, fără să „stricați” lanțul.

5.3 Legea Demeter („vorbește doar cu vecinii tăi”)

• Nu amestecați „detalii de format” peste tot. Dacă JSONul își schimbă schema, ideal este să modificați un singur adaptor (normalizeRecords), nu toată aplicația.

6. Robusteză operațională — edge‑cases și politici

6.1 Liste goale și absența init‑ului

• reduce fără init pe [] este eroare; cu init, definești cazul de bază. Introduceți invarianta „reduce are întotdeauna init” în bibliotecile interne; testele vă ajută să o păstrați.

6.2 Date invalide / lipsă

• Scrieți predicate clare: isValidRecord — verifică name, faculty ∈ set permis, interests listă nenulă. E mai bine să respingi cu motiv (log) decât să accepți tăcut. În laboratorul practic vom separa „acceptate”/„respinse” prin partition.

6.3 Duplicări

• uniqBy(emailNormalize). Acceptați că e‑mailul nu este un identificator perfect (forwarding, aliasuri), dar pentru laborator este suficient.

6.4 Comparator determinist

• Evitați rezultate „oscillante” la egalități prin al doilea/ al treilea criteriu. Determinismul simplifică testarea și reproducerea.

6.5 Numeric vs. lexical

• "10" < "2" este true lexical; convertiți la număr. Pentru date, preferați un format canonic (ISO‑8601), altfel comparațiile devin ambigue.

6.6 Localizare/diacritice

• Pentru ordonare „umană” pe text românesc/internațional, folosiți Intl.Collator; notați costul în performanță și testați pe seturi reprezentative.

7. Performanță: Big‑O pragmatic, memorie, „fusion” și când să optimizăm

7.1 Ordine de mărime

• map/filter: O(n); reduce: O(n); sort: O(n log n). Intersection/difference cu Set: O(n + m).

• Memoria: fiecare etapă produce uneori colecții intermediare. Fusion (ex., combinarea a două map‑uri într‑unul singur) poate reduce aceste colecții, dar complica debuggingul.

7.2 Pre‑optimizarea este sursa a 80% din probleme

• În material educațional, întâi claritate + teste verzi. Apoi profilare (console.time/timeEnd sau benchmarkuri simple) și optimizări măsurate (nu „pe suspiciuni”).

7.3 Legătura cu MapReduce (nivel conceptual)

• MapReduce ne amintește că map + reduce sunt o „schemă” robustă pentru prelucrări masive: map produce perechi intermediare (cheie, valoare), reduce le grupează după cheie și agregă. În codul local, analogia rămâne: map proiectează/normalizează, reduce agregă; grupările (groupBy/countBy) reproduc schema la scară mică.

8. Corectitudine: semantică, istorie și capcane JS

8.1 De ce apar „surprize” în JS și cum le evităm

• Istoric, JavaScript a evoluat rapid pentru a răspunde nevoilor web‑ului. Unele decizii au compromis simplicitatea în favoarea compatibilității (coerciuni, this). Documentele istorice arată cum comunitatea a standardizat gradual practicile mai robuste, ceea ce explică de ce astăzi recomandăm arrow functions, const, modele declarative și separarea logicii pure (vezi bibliografia pentru o perspectivă istorică).

• Forme de „coerciune surprinzătoare”: [] + {} vs. {} + []; comparații == vs. ===; adevăr/fals implicite. Soluția pragmatică: ===, conversii explicite și utilitare de normalizare.

8.2 Semantici executabile și verificare

• Există lucrări care formalizează complet semantica JS (de ex., KJS), utile pentru unelte, verificări și comparații între implementări. Consecința pentru noi: cu cât codul este mai „pur” și contractual, cu atât e mai ușor de verificat, testat și explicat.

8.3 Standardizare și ecosisteme

• ESM (ECMAScript Modules) oferă import/export predictibil; separarea modulelor ajută la testare. Evitați amestecul ESM/CJS într‑același modul.

• Tooling: în laborator folosim numai Node pur și test‑runnere (Vitest/Jest) — suficient pentru a înțelege principiile. În proiectele mari, veți completa cu bundlere, transpileri, lintere și tipare (TypeScript).

9. Testare și calitate — cum scriem contracte utile

9.1 Zero‑unu‑mulți

• Pentru fiecare utilitar, testați pe liste goale (zero), liste cu un element (unu) și liste cu mai multe (mulți). Această triadă surprinde o mare clasă de bug‑uri.

9.2 Edge‑cases curate

• Ferestre de invalidare: elemente lipsă, duplicate, tipuri combinate, diacritice, spații.

• Invarianta de contorizare: la countBy, suma frecvențelor = lungimea listei „normalizate”.

9.3 Vitest & Jest „în oglindă”

• Rularea acelorași contracte în ambele suite are valoare didactică: deprinzi reflexe transferabile (matcher‑e, structuri de fișiere, configurări). Nu dublezi munca: copiind contractele, validezi ecosisteme diferite.

9.4 Property‑based (idee, fără biblioteci) și teste metamorfice

• Generați manual 20–30 de intrări pseudo‑aleatoare și verificați invarianta (de ex., uniqBy păstrează doar primul/ultimul după politica declarată). Metamorfism: dacă sortezi cu comparator stabil și apoi reordonezi egalitățile alfabetic, rezultatul rămâne stabil.

9.5 Linting & naming

• ESLint (flat config) pentru reguli de bază: no‑undef, no‑unused‑vars (relaxat pentru argumente nefolosite), sourceType module. Denumiri explicite: groupByFacultyYear, toFrequencyEntries, byCountThenAlpha. Comentarii scurte, orientate pe „intenție”, nu pe „ce face fiecare linie”.

10. AI‑assist (Copilot sub VSL / ChatGPT/Mistral/Claude) — cum grăbim fără să stricăm

10.1 Strategia VSL

• Verify (Tests) — cere mai întâi teste. Ex.: „Generează 10 teste pentru countBy pe intrări cu interese multiple pe rând, incluzând cazuri goale și duplicate.”

• Specify (Contracts) — validează semnăturile și politica (chei duplicate? lowercase?).

• Learn/Lint — cere un refactor „păstrând contractul” sau explică schimbările sugerate.

10.2 Prompting eficient

• „Scrie map/filter/reduce pentru top‑K interese, cu comparator stabil (desc pe count, apoi alfa). Explică în 3 propoziții de ce comparatorul e stabil.”

• „Dă‑mi 8 edge‑cases realiste pentru normalizarea intereselor (delimitatori diferiți, whitespace, cazuri lipsă).”

• „Rescrie keyBy pentru politică ‘last wins’ și demonstrează prin 3 teste.”

10.3 Garduri de protecție

• Nu lipi cod fără test. Dacă AI oferă un comparator scurt care „pare” corect, scrie 4 teste contraintuitive (egalități, diacritice, mix numeric+text).

• Cere întotdeauna un „diff semantic”: ce schimbă propunerea asupra complexității? asupra politicilor? asupra mutabilității?

10.4 Colaborare în echipă

• Stabiliți un set de „contracte” într‑un README. AI e mai eficient când promtul include aceste reguli; altfel, va propune stiluri inconsistente.

11. Studii de caz scurte (fără cod sau cu pseudo‑cod; detaliem în Partea 2)

11.1 Top‑K interese (cu normalizare)

• Intrări: liste de rânduri cu interests divers formate.

• Pași: flatMap → normalize → filter(valid) → countBy → entries → sort(byCountThenAlpha) → slice(0,K).

• Verificări: suma frecvențelor = număr de interese valide; ordinea stabilă pentru egalități.

11.2 Distribuție pe (facultate, an)

• Cheie compusă; groupBy → map către pair (facultate, an) → sort și formate pentru raport.

11.3 Deduplicare înscrieri

• uniqBy(emailCanon); raportezi câte ai eliminat și de ce (log).

11.4 Partition valid/invalid + motivare

• partition(isValid); din invalid, extrage motivele (lipsă faculty, interes gol etc.); contorizează după motiv pentru feedback către autorii formularului.

11.5 „Audit” prin tap/peek

• Inserarea de „tap” după fiecare pas pentru a verifica dimensiunea curentă a listei. În testele de integrare, tap devine un loc în care aserți „la jumătatea pipeline‑ului avem N elemente”.

12. Arhitectură de bibliotecă — separare de responsabilități

• src/lib/normalize.js — normalizează câmpuri (faculty, interests, email).

• src/lib/collect.js — groupBy, countBy, keyBy etc.

• src/lib/predicates.js — byFaculty(name), hasInterest(x), isValid(record).

• src/lib/sorters.js — byCountThenAlpha, byFacultyThenYear.

• src/index.js — fațadă care re‑exportă utilitarele; facilitează testarea și schimbarea structurii interne.

• Separarea oferă flexibilitate: dacă decizi să normalizezi altfel (e.g., diacritice), modifici doar normalize.js și testele aferente; restul rămâne.

13. Compoziții frecvente („rețete” re‑utilizabile)

• buildTopKInterests(K) — întoarce o funcție care, aplicată pe lista de rânduri, produce top‑K.

• buildFacultyYearDistribution() — produce un tabel agregat pe (facultate, an).

• buildDedupReport() — întoarce {accepted, rejected, reasonsCount}.

• buildUserFacingSummary(format = 'table' | 'json') — serializare a rezultatelor.

14. Considerații despre date reale — igienă, etică, reproducibilitate

• Igienă: logați motivele de respingere; evitați să aruncați în tăcere.

• Etică: nu includeți date sensibile în rapoarte publice; anonimizare când este cazul.

• Reproducibilitate: fixați versiuni de dependențe; salvați „fixtures” reprezentative; includeți un README cu pașii de re‑rulare a rapoartelor.

15. Privire istorică și epistemologică (de ce a ajuns JS aici)

• În literatura istorică găsiți o articulare coerentă a deciziilor care au conturat JS. Acea cronologie explică de ce avem astăzi un ecosistem în care „programarea pe colecții” e naturală (metode integrate, ESM), dar și de ce persistă capcane de coercion/this. Înțelegerea trecutului vă ferește de polemici sterile și vă îndrumă către practici solide: funcții pure, testare, compoziție declarativă, contracte explicite. În laborator veți transpune această filozofie în cod concret și verificabil.

16. Concluzie

Am pus accentul pe gândirea „în transformări”: fiecare pas are o intenție clară și produce o structură intermediară predictibilă. map/flatMap/filter/reduce sunt verigile principale; utilitarele (groupBy, countBy, keyBy etc.) sunt „expresii” mai bogate care reduc ambiguitatea și cresc lizibilitatea. Calitatea nu vine din „șmecherii” sintactice, ci din contracte clare, testate și respectate. În Partea 2 (Laborator) vei implementa o mini‑bibliotecă, vei construi un pipeline complet pentru rapoarte și vei rula aceleași contracte în Vitest și în Jest pentru a fixa deprinderile. În Partea 3 vei avea 45 de proiecte graduale (L1/L2/L3) pentru a consolida aceste practici în contexte ușor diferite.

Note bibliografice (APA 7, cu DOI real)

• Hughes, J. (1989). Why functional programming matters. The Computer Journal, 32(2), 98–107. https://doi.org/10.1093/comjnl/32.2.98

• Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. Communications of the ACM, 51(1), 107–113. https://doi.org/10.1145/1327452.1327492

• Park, D., Ștefănescu, A., & Roșu, G. (2015). KJS: A complete formal semantics of JavaScript. ACM SIGPLAN Notices (PLDI), 50(6), 346–356. https://doi.org/10.1145/2737924.2737991

• Ryu, S., & Park, J. (2024). JavaScript language design and implementation in tandem. Communications of the ACM. https://doi.org/10.1145/3624723

• Wirfs‑Brock, A., & Eich, B. (2020). JavaScript: The first 20 years. Proceedings of the ACM on Programming Languages, 4(HOPL), Article 77, 1–189. https://doi.org/10.1145/3386327

17. Tipare avansate pe array‑uri — dincolo de bază

17.1 Ferestre glisante (sliding window)

• Anumite analize cer contexte locale (ex.: medie mobilă pe un scor de participare). Deși nu e subiectul central al acestui seminar, e util să înțelegi cum le‑ai construi declarativ: definești o funcție windowed(n) care transformă [a,b,c,d,e] în [[a,b,c],[b,c,d],[c,d,e]] și apoi aplici map pe fiecare fereastră. Complexitate: O(n), dar cost de memorie pentru ferestre. În laboratorul extins din seminare viitoare poți optimiza prin generatoare.

17.2 Reconciliere / „join” între liste

• În date reale, vrei să îmbogățești rânduri (ex.: asociază fiecărui student un „major” sau un „score” dintr‑o altă listă). În SQL ai JOIN; în JS poți simula „inner join”/„left join” folosind keyBy pe lista de referință și apoi map pe lista principală. Clarifică politica la chei lipsă: pui null, arunci eroare, sau marchezi pentru „follow‑up”?

17.3 Fuzionarea a două liste ordonate (merge)

• Dacă ai două liste deja ordonate (ex.: participanți ordonați după dată și sponsori ordonați după buget), poți face un „merge” linear (O(n+m)). Acest tipar merită cunoscut când vrei să eviți sortarea repetată a reuniunilor de liste.

17.4 Agregări multiple într‑o singură trecere

• reduce poate colecta simultan mai multe statistici: {count, sum, min, max, byFaculty: Map, byInterest: Map}. Avantaj: o singură trecere O(n). Dezavantaj: acumulatoare mai „grele” și mai greu de testat dacă nu păstrezi claritatea contractului. Compromis: extrage sub‑agregări în funcții mici (incCountByFaculty(acc, row)).

17.5 Transformări „lossless” vs. „lossy”

• Unele transformări păstrează toate informațiile (map proiectiv, sort, groupBy dacă păstrezi listele). Altele „pierd” detalii (countBy, uniq). E util să marchezi în README când pipeline‑ul devine ireversibil (nu mai poți reconstrui intrarea din ieșire).

18. Tratamente pentru erori — politici explicite

18.1 Aruncare controlată vs. semnalare

• În utilitare de bază, e preferat să nu arunci excepții pentru cazuri de date „obișnuite” (lipsă facultate) ci să semnalezi prin return (ex.: {ok:false, reason:'NO\_FACULTY'}). Excepțiile păstrează‑le pentru cazuri de programare (ex.: comparator invalid).

• În pipeline‑uri, un stil robust combină partition(isValid) cu raportarea motivelor (statistică pe „reasons”).

18.2 Valorile „sentinelă”

• Poți folosi null/undefined ca marcaje temporare, dar evită să le lași „să curgă” în pașii următori fără filtru. Teoria: fiecare etapă ar trebui să „consume” sentinelul (prin filter/compact) sau să‑l convertească într‑o formă explicită (Either‑like).

18.3 Idempotență

• Dacă rerulezi pipeline‑ul pe aceleași date, ar trebui să obții același rezultat. Lipsa idempotenței e un semn că ai efecte secundare ascunse (ex.: comparator care modifică obiectele). Testează idempotența în suitele tale (execută de 3 ori și compară).

19. JSON, timp și numere — note de atenționare

19.1 JSON și tipurile sale limitate

• JSON nu cunoaște Date sau Map/Set; serializarea/parse‑ul poate „aplatiza” structuri. În utilitare, lucrează cu structuri „JSON‑compatibile” (obiecte, array‑uri, stringuri, numere), iar pentru Date adoptă formatul ISO‑8601 (UTC) ca „limbă comună”.

19.2 Numerele în virgula flotantă

• În agregări numerice, erorile de reprezentare pot produce surprize (0.1 + 0.2 ≠ 0.3). Pentru rapoarte statistice simple, e suficient să formatezi la un număr de zecimale fix. Pentru calcule sensibile (ex.: bugete), învață ulterior să folosești BigInt sau biblioteci decimale.

19.3 Time‑zones și comparări de dată

• Compară datele în format ISO‑8601 zecimal sau convertește explicit la milisecunde UTC. Evită „string compare” pe date locale.

20. Generator functions (function\*) și iterație „lazy”

20.1 De ce lazy?

• Când fișierele devin mari sau când vrei „streaming”, nu vrei colecții intermediare mari. Prin generatoare, poți replica map/filter și la nivel de iteratoare (ex.: mapGen, filterGen, take). În acest seminar rămânem pe array‑uri pentru claritate, dar e util să știi direcția de evoluție pentru seturi mari.

20.2 Interoperabilitate

• Poți converti ușor între array și iterator: Array.from(iterator) sau yield\* array în generator. Decizia ține de memorie și de ergonomie.

21. Organizarea codului — module, fațade, puncte de extensie

21.1 Module ESM

• Re‑exportă dintr‑un index.js pentru a controla API‑ul public. În teste, importă din index (nu din module interne), ca să nu „bypassezi” contractul public.

21.2 Puncte de extensie

• Pentru predicatele frecvente, construiește generatoare de predicate: byFaculty('CSIE'), hasInterest('AI'). Astfel, compui predicate (AND/OR) fără a crește complexitatea funcțiilor de bază.

22. Documentare și comunicare

22.1 README ca contract

• Include scop, convenții (lowercase pentru e‑mailuri), politici la duplicate, exemple de rulare și secțiune „Edge cases”. Un coleg nou va „înțelege API‑ul” în 5 minute.

22.2 Exemple minim‑suficiente

• Evită romane în README; două exemple bine alese (fericit + edge) sunt mai valoroase decât zece superficiale.

23. Micro‑studiu de caz: „StudentHub Insights” (descriere exhaustivă fără cod)

23.1 Context

• Date: 1 200 înscrieri, 4 facultăți, 6 interese principale, 3 ani de studiu; surse: CSV (formular vechi), JSON (formular nou).

23.2 Obiective

• top‑K interese (K=5), distribuții pe (facultate, an), deduplicare pe e‑mail, raportare a motivelor „invalid”.

23.3 Politici

• faculty ∈ {CSIE,FABBV,REI,MAN}; interests normalizat din string/array; e‑mail case‑insensitive, spații eliminate.

23.4 Pipeline narativ

• Normalizare: transformă fiecare rând (trim, lowercase, split/flatMap interese).

• Filtrare: reține rânduri cu faculty validă și cel puțin un interes.

• Deduplicare: uniqBy(emailCanon).

• Agregare: countBy pentru interese; groupBy pe (faculty, year) pentru distribuții.

• Ordonare: comparator byCountThenAlpha; tie‑break explicit.

• Raportare: două formate (table/json) cu aceleași date.

23.5 Validări

• Invarianta sumă frecvențe = #interese după normalizare.

• Stabilitate la ordonare (egalități).

• Independență între pași (deduplicarea nu trebuie să modifice interesele normalizate).

23.6 Rezultat narativ

• Managerul primește top‑K și distribuția; poate decide 2 ateliere AI, 1 Web, 1 UX, cu focus pe anii I–II la CSIE și REI.

23.7 Post‑mortem

• Două clase de erori frecvente: date „jumătate completate” (interes lipsă) și dubluri (același e‑mail). Ajustăm formularul (validare client) și adăugăm feedback în confirmare.

24. Practici de echipă — cum păstrezi calitatea în timp

24.1 Revizuire pe bază de exemple

• Un PR care introduce un utilitar nou trebuie să adauge 4–6 teste reprezentative. Reviewerii se uită mai întâi la teste, abia apoi la implementare.

24.2 „Golden files”

• Pentru rapoarte JSON, păstrează un fișier „golden” mic în repo. Testele compară output‑ul actual cu golden; diferențele sunt intenționate sau bug‑uri.

24.3 Semver intern

• Chiar dacă nu publici pe npm, tratează API‑ul intern cu grijă: schimbările de contract deserve release‑uri „major” interne și note în CHANGELOG.

25. Antipattern‑uri și remedii

25.1 „Reducer‑ul bag‑de‑toate”

• reduce în care bagi orice logică devine opac. Rupe în utilitare mici (incCount, appendGroup) și testează separat.

25.2 „Comparatorul magic”

• Comparatori care fac și normalizare, și logare, și side‑effects. Păstrează comparatorul pur; normalizarea se face în preprocesare.

25.3 „Helperi” cu efecte ascunse

• Funcții care „par” pure dar setează variabile în afara lor. Interzise în bibliotecile de utilitare; efectele se fac la margini (I/O).

26. Seturi de prompturi AI „gata de folosit” (română/engleză)

• „Generează 12 teste pentru groupBy(key) care acoperă: chei lipsă, chei compuse, liste goale, distribuții dezechilibrate, elemente null.”

• „Rewrite the comparator to be stable with a secondary alphabetical tie‑break; explain in 3 sentences and keep O(n log n) complexity.”

• „Propune 10 edge cases for uniqBy(email) including casefolding, accents stripping, whitespace, nulls, and repeated aliases.”

• „Create a refactor plan to split a monolithic reduce into small pure helpers while preserving the external contract.”

• „Suggest property‑based test ideas (without libraries) for countBy; show how to randomize inputs and assert the conservation law.”

27. Legături cu cercetarea și standardizarea (context academic)

• De ce funcționalele contează: modul în care higher‑order functions și (în limbaje lazy) evaluarea leneșă cresc modularitatea și permit exprimarea elegantă a agregărilor. Lectura contribuie la dezvoltarea unei intuiții solide despre compoziție și testabilitate.

• De ce map/reduce a devenit „limbaj comun” pentru prelucrări mari: modelul conceptual s‑a dus de la teorie la implementări pe clustere; pentru noi, ecoul practic este proiectarea clară a pașilor și a contractelor.

• De ce formalizările semantice contează: unelte fiabile și încredere în comportament; chiar dacă nu „scriem” semantici, respectăm principiile—funcții pure și contracte—care se verifică și formal.

28. Pregătirea pentru Partea 2 (Laborator)

• Vei implementa utilitare (groupBy, countBy, keyBy, partition, uniqBy, difference, intersection, chunk, zip, flatten) cu teste în oglindă (Vitest/Jest).

• Vei construi un pipeline map/filter/reduce parametrizabil (K, predicate compuse) și vei exporta un raport în două formate.

• Vei documenta politicile în README și vei scrie un set de prompturi AI‑assist „test‑first” pentru a accelera munca.

• Obiectivul nu este „să știi pe de rost toate metodele”, ci să poți proiecta repede, corect și lizibil o transformare de date cu contracte verificabile.