**Seminarul 1 — Introducere în JavaScript: sintaxă modernă, funcții, iterație, CLI  
(Partea 1 — Teorie, narativă, hook realist, exemple și subcapitole)**

\*\*Hook realist: „Raport în 45 de minute”\*\*

Este ora 08:45 într-o zi de marți și ești voluntar tehnic în Asociația Studențească „StudentHub”. La 09:30 are loc o ședință scurtă cu Biroul Executiv, iar președintele îți trimite în grabă un fișier `inscrieri.csv` exportat dintr-un formular online. Ținta: un sumar curat al înscrierilor — \*total candidați\*, \*distribuție pe facultăți\*, \*top 5 interese\* — și o listă filtrată cu „membri noi”. Instalații disponibile: doar \*\*Node.js\*\* pe un laptop simplu. Nu ai voie să instalezi aplicații complexe sau să exporți datele către alte servicii. În 45 de minute, trebuie să livrezi un mic \*\*utilitar CLI (command‑line interface)\*\* care citește fișierul, curăță datele și produce un raport textual. De aici începe mersul nostru: \*\*JS modern (ES)\*\*, gândire funcțională pentru transformări sigure, iterație expresivă, \*error handling\* clar și o disciplină de calitate demonstrabilă.

---

## 1. Ce este JavaScript astăzi (context, standardizare, runtime‑uri)

\*\*JavaScript\*\* este limbajul de facto pentru logică în browser, dar, prin \*\*Node.js\*\*, a devenit și o unealtă matură de \*\*automare (automation)\*\*, \*\*scripting\*\* și \*\*CLI\*\* cross‑platform. Limbajul este standardizat anual sub \*\*ECMAScript (ECMA‑262)\*\* de către Ecma International (\*TC39\*), iar ritmul incremental (ediție în fiecare iunie) a fixat un nucleu modern: \*block scoping\* (`let`, `const`), \*arrow functions\*, destructurare, \*rest/spread\*, \*template literals\*, colecții noi (\*Map/Set\*), \*modules (ESM)\* și diverse rafinamente semantice. Standardizarea a evoluat istoric din nevoia de interoperabilitate între motoare independente de browser; în paralel, comunitatea de cercetare a urmărit să descrie formal semantica limbajului, de exemplu prin lucrări precum \*\*KJS\*\* — o semantică completă a ECMAScript 5.1 testată extensiv pe suitele de conformanță (Park, Ștefănescu & Roșu, 2015). citeturn6search3turn0search4

\*\*Browser vs. Node\*\*. Același limbaj, dar \*\*ambient diferit\*\*: în browser ai \*\*DOM\*\*, \*Web APIs\* (de pildă, `fetch`, `localStorage`), ciclu de viață al paginii și restricții de securitate (CSP, \*same‑origin policy\*). În \*\*Node\*\*, nu ai DOM nativ; în schimb ai acces la \*\*sistemul de fișiere\*\*, \*\*rețea\*\*, \*\*process.env\*\*, \*\*process.argv\*\* și posibilitatea de a publica utilitare CLI. Practic, vei scrie funcții pure (transformare date) \*independente de ambient\* și, la margini, adaptoare pentru \*\*I/O\*\*.

> \*Note pedagogic\*: în această parte ne concentrăm pe \*\*limbaj\*\* și pe abilitățile de a scrie \*\*funcții compoziționale\*\* și \*\*iterații declarative\*\*; în Partea 2 transformăm aceste cunoștințe într‑un utilitar CLI real pentru scenariul „StudentHub”.

---

## 2. Tipuri, valori și conversii (coercion) — de ce predictibilitatea contează

JS are tipuri primare (`string`, `number`, `boolean`, `bigint`, `symbol`, `null`, `undefined`) și agregate (`object`, `function`, `array` ca obiect specializat). Datorită naturii \*\*dinamice\*\* a limbajului, conversiile implicite (coercion) pot produce surprize (de pildă, `[] + {}` vs. `{} + []`). \*\*Principii practice\*\*:

- Preferă \*\*conversii explicite\*\*: `Number(x)`, `String(x)`, `Boolean(x)` în loc de rely pe „adevărul” JS în expresii logice.

- Evită „presupuneri” despre \*truthiness\* în calcule critice; dacă intrarea poate lipsi, normalizeaz-o cu \*\*nullish coalescing\*\* (`??`) și evaluează explicit cazurile `null`/`undefined`.

- Tratează `NaN` ca \*\*valoare contaminantă\*\*: se propagă prin aritmetică. Verifică prin `Number.isNaN`.

> \*Comparatia din viața reală\*: dacă tipurile sunt „formatul ambalajului”, conversiile implicite sunt ca reetichetarea automată în depozit — poate funcționa, dar riști să expediezi o cutie „Fragil” ca „Greutate” fără să vrei. În procese critice (rapoarte, bani, note), \*\*etichetarea manuală\*\* (conversii explicite) e standardul de aur.

---

## 3. Variabile, domenii (scope) și memoria conceptuală

\*\*`let` și `const`\*\* introduc \*block scoping\*, eliminând multe capcane istorice ale lui `var`. `const` indică \*\*binding imutabil\*\* (nu neapărat valoare imutabilă), iar `let` permite reatribuirea. \*Temporal Dead Zone (TDZ)\* explică de ce folosirea unei variabile \*\*înainte\*\* de declarare produce `ReferenceError` — un mecanism care previne o clasă întreagă de bug‑uri subtile.

- Evită reatribuirea pentru „valori liant” (identități), preferând `const` pentru claritate.

- Grupează variabilele după \*\*răspundere\*\*; limitează vizibilitatea (scope mic → mai puține șanse de coliziune).

\*\*`this` și funcțiile săgeată\*\*. `this` este \*\*dinamic\*\* în funcțiile obișnuite (depinde de apelant), dar \*\*lexical\*\* în \*arrow functions\* (moștenește contextul). În logică pură, evită dependențele de `this`; preferă parametri expliciți.

> \*Analogie\*: `scope` este „camera” în care lași instrumentele. Dacă ai o singură sală uriașă (global), toate sculele se amestecă. Camerele mici (blocuri) reduc riscul să calci pe o unealtă nepotrivită.

---

## 4. Funcții moderne: declarații, expresii, \*arrow\*, \*default/rest/spread\*, compoziție

Funcțiile sunt atomi principali în JS. Câteva idiomuri practice:

- \*\*Declarație\*\*: ridicată (hoisted) în cadrul blocului/scope‑ului.

- \*\*Expresie\*\*: atribuită unei variabile; control mai fin asupra vizibilității.

- \*\*Arrow\*\*: concisă, \*lexical this\*, utilă în transformări; fii atent la \*\*return\*\* implicit (fără acolade) vs. bloc (cu acolade + `return`).

\*\*Parametri moderni\*\*:

- \*Default parameters\*: specifică valori implicite robuste (`function f(x = 0) { ... }`).

- \*Rest (`...args`)\*: colectează argumente variadice într‑un array.

- \*Spread (`...array`)\*: dezasamblează o colecție în elemente, util la clonări superficiale și combinări.

\*\*Compoziție și funcții de ordin superior (HOFs)\*\*. „Funcția care primește/returnează funcție” permite \*\*abstracții compacte\*\*: \*filters\*, \*mappers\*, \*reducers\*. Beneficiile programării funcționale — modularitate, reutilizare, compoziție — sunt argumentate clasic de Hughes (1989) și rămân actuale: cod mai scurt, cu \*\*depedențe locale și efecte controlate\*\*. citeturn0search11

> \*Heuristică\*: Scrie funcțiile „ca transformări” — de la \*\*input\*\* clar la \*\*output\*\* clar — și amână efectele (I/O) către margini. Asta face codul \*\*ușor de testat\*\* și \*\*combinabil\*\*.

---

## 5. Iterație: imperative (for/while) vs. declarativ (map/filter/reduce), iterabile & generatoare

\*\*Bucla\*\* rămâne utilă pentru proceduri pas cu pas, dar în transformări de date, \*\*stilul declarativ\*\* domină:

- `map`: transformă element cu element (păstrează lungimea).

- `filter`: păstrează elementele care trec un predicat.

- `reduce`: „pliază” o listă într‑o singură valoare (sau structură).

Aceste \*pipelines\* reflectă idei mai vechi din procesarea de date la scară („\*\*MapReduce\*\*” — desigur, într‑un mediu total diferit față de array‑urile din JS, dar cu o \*\*intuiție\*\* comună: \*map\* pentru producerea perechilor intermediare, \*reduce\* pentru agregare) (Dean & Ghemawat, 2008). citeturn0search2

\*\*Când `for`?\*\* Când ai nevoie de \*\*scurtcircuitări\*\* cu \*break/continue\*, când micro‑optimizarea locală chiar contează, sau când transformarea nu se potrivește natural pe `map/filter/reduce`.

\*\*Iterabile & generatori\*\*. \*Protocolul iterabil\* vizează structuri care expun un \*iterator\* (`[Symbol.iterator]`). \*Generatorii\* (`function\*`) creează \*\*secuențe leneșe (lazy)\*\* cu `yield` — utile pentru fluxuri mari sau potențial infinite.

> \*Analogie\*: `map/filter/reduce` sunt „stații” pe o \*\*linie de asamblare\*\*. Fiecare stație are un rol clar; rezultatul final iese la capăt fără să te oprești să „demontezi” produsul între stații.

---

## 6. Destructurare, \*pattern matching\* simplu și obiecte

\*\*Destructurarea\*\* („spargerea” structurată) ajută la extragerea câmpurilor/elementelor concrete din obiecte/array‑uri, făcând codul \*\*declarativ\*\*. Exemple: extragerea `name` și `faculty` dintr‑un `student`, sau `first, ...rest` dintr‑o listă. Combinată cu \*default values\* și \*renaming\*, reduce \*boilerplate\*.

\*\*Obiecte\*\*: în JS, obiectele sunt \*\*dicționare dinamice\*\* care pot fi prototipate (lanțuri de moștenire prin `[[Prototype]]`). În acest seminar ne oprim la utilizarea de bază (creare literală, proprietăți, metode scurte, \*computed keys\*); OOP, prototipuri și \*class\* vor fi tratate sistematic în seminariile ulterioare.

---

## 7. Erori și \*error handling\*

Erorile sunt inevitabile în prelucrarea datelor. \*\*Model recomandat\*\*:

- \*\*Validare\*\* timpurie la intrare (tipuri așteptate, câmpuri obligatorii).

- \*\*Semnalare\*\* concisă: aruncă `Error` cu mesaj clar; folosește `try/catch` doar unde poți \*\*gestiona\*\* (nu „ascunde”).

- \*\*Coduri de ieșire\*\* semnificative în CLI (`process.exitCode = 1`).

\*\*Capcane\*\*: a nu interpreta corect `JSON.parse` vs. string malformat; a nu izola I/O de calcul (teste fragile). \*Guardrails\*: teste unitare pe funcțiile pure; teste de integrare minime pe I/O.

---

## 8. CLI în Node.js: \*process.argv\*, \*process.env\*, I/O și norme POSIX

Un \*\*CLI\*\* robust înseamnă \*\*contract\*\*: ce argumente acceptă, ce iese pe `stdout`, ce erori apar pe `stderr`, ce cod de ieșire semnifică reușită/eșec. Tradiția POSIX a riguros stabilit comportamente pentru utilitare (opțiuni scurte/lungi, \*exit codes\*, \*stderr\* pentru erori), conturând un \*ux\* al liniei de comandă adoptat pe scară largă (Walli, 1995). citeturn2search1

\*\*Elemente practice\*\*:

- `process.argv` → parsezi argumentele (poți începe fără dependențe, apoi, când devine complex, apelezi la \*yargs/commander\*).

- `process.env` → configurare fără hard‑coding (ex.: `REPORT\_LIMIT=5`).

- \*\*I/O fișiere\*\* → \*fs/promises\* pentru asincron; dar logica pură să rămână testabilă separat.

- \*\*Mesaje\*\* și \*\*coduri de ieșire\*\* previzibile: `0` (ok), non‑zero (erori clasificate).

> \*Analogie\*: un CLI e ca un \*\*automat\*\* de bilete la gară: butoane clare (opțiuni), chitanță lizibilă (raport), și lumina roșie dacă nu ai introdus suma corectă (cod de ieșire ≠ 0).

---

## 9. Rigoare și corectitudine: ce înseamnă „să respecți specificația”

Deși scriem utilitare mici, merită să cunoști efortul comunității pentru \*\*specificații precise\*\* și \*\*verificări\*\*. „KJS” a oferit o semantică executabilă care trece mii de teste standardizate; mai recent, eforturi de vizualizare și mecanizare a ECMA‑262 arată cum limbajul și implementările pot co‑evolua cu \*\*testare diferențială\*\* și \*type‑checking\* al specificației (Park et al., 2015; Choe et al., 2025; Ryu & Park, 2024). Concluzia practică pentru noi: scrie \*\*contracte\*\* (tipuri așteptate, forme de date) și \*\*teste\*\* simple — vei câștiga predictibilitate. citeturn0search4turn6search0turn7view0

---

## 10. Stil, \*linting\*, testabilitate

Un stil consecvent (nume descriptive, funcții mici, un singur nivel de abstracție per funcție) reduce costul cognitiv. \*\*Linting\*\* modern (flat config) și \*\*formatting\*\* automat pun ordinea înaintea dezbaterilor despre spații. Din perspectiva testelor:

- \*\*Teste pe „unități pure”\*\* (ex.: `parseRow`, `groupByFaculty`, `topInterests`): ușoare, rapide, stabile.

- \*\*Oglindă Jest/Vitest\*\*: aceleași \*contracte\*, două ecosisteme populare → familiaritate transferabilă.

---

## 11. Cum folosim ajutorul AI (Copilot sub VSL, ChatGPT/Mistral/Claude) fără a compromite rigoarea

\*\*Principiu\*\*: AI ca „co‑pilot”, \*\*nu\*\* ca „autoritate” — \*ghost text\* pentru schițe, noi decidem contractele. Studiile recente arată limite obiective ale LLM‑urilor în identificarea și raționarea pe vulnerabilități; ieșirile pot fi \*\*nedeterministe\*\* și uneori \*\*eronate\*\* (Ullah et al., 2023). De aceea:

1. \*\*Îi dictezi contractul\*\* (\*signature\*, \*pre/postconditions\*, \*edge‑cases\*).

2. \*\*Ceri teste\*\* înainte de implementare („test‑first prompts”).

3. \*\*Refactorezi\*\* împreună, dar \*\*verifici\*\* cu testele tale.

4. \*\*Nu delegi decizii de securitate\*\* către AI; folosești doar ca sursă de idei, nu de verdict. citeturn3search14

\*\*Exemple de \*prompts\* eficiente (în română, termeni tehnici în English):\*\*

- „Scrie o \*\*pure function\*\* `groupByFaculty(students: Array<Student>)` care returnează `Record<string, number>`. Include \*\*3 edge‑cases\*\*: `[]`, `faculty` lipsă, `faculty` non‑string. Nu folosi I/O. Afișează \*\*doar\*\* codul.”

- „Compune un \*\*pipeline (map → filter → reduce)\*\* pentru a extrage `top 5 interests` dintr‑un array de obiecte `{ interests: string[] }`. Adaugă \*\*comentarii succinte\*\* la fiecare pas.”

- „Propune un \*\*refactor\*\* pentru `parseCsvLine(line: string)` astfel încât să trateze \*separatorul\* și \*trim\* configurabile. Menține semnătura și \*\*adaugă 4 teste\*\* (Vitest).”

---

## 12. Mini‑exemple de cod (doar ca ancore pentru concepte)

\*\*Destructurare & valori implicite\*\*:

const student = { name: "Ana", faculty: "CSIE" }

const { name, faculty, group = "N/A" } = student

\*\*Compoziție declarativă\*\* (top interese):

const topInterests = (rows, k = 5) =>

Object.entries(

rows

.flatMap(r => r.interests ?? [])

.reduce((acc, it) => (acc[it] = (acc[it] ?? 0) + 1, acc), {})

)

.sort((a, b) => b[1] - a[1])

.slice(0, k)

\*\*CLI: argumente și cod de ieșire\*\*:

const args = process.argv.slice(2)

if (args.length < 1) {

console.error("Usage: node cli.js <path-to-csv>")

process.exitCode = 1

}

> \*Reamintire\*: aceste fragmente sunt doar ancore; laboratorul din Partea 2 dezvoltă un \*\*mini‑CLI\*\* complet testat (Vitest & Jest).

---

## 13. Concluzie & punte către laborator

Ai văzut „piesele de Lego” ale JS modern: \*\*tipuri și conversii explicite\*\*, \*\*`let/const`\*\* și \*\*TDZ\*\* pentru disciplină, \*\*funcții\*\* ca atomi de compoziție, \*\*iterații declarative\*\* pentru transformări robust testabile, \*\*generatori\*\* ca instrument avansat, \*\*erori\*\* gestionate clar și \*\*CLI\*\* ca țintă practică. În laborator vei pleca de la \*\*micro‑exerciții pure\*\* (ex.: `parseRow`, `validate`, `groupBy`) și vei ajunge la un \*\*utilitar\*\* care produce raportul cerut în \*\*scenariul StudentHub\*\*. Pe tot parcursul, \*\*AI‑assist\*\* îți scurtează drumul, însă \*\*testele\*\* rămân arbitrii finali.

---

## Bibliografie (APA 7th)

- Choe, M., Song, K., Kim, H., & Park, J. (2025). \*A JavaScript language specification visualization tool\* (JSSpecVis). In \*\*Proceedings of the 33rd ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE 2025)\*\*. https://doi.org/10.1145/3696630.3728579

- 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

- Ducasse, S., Petton, N., Polito, G., & Cassou, D. (2012). \*Semantics and security issues in JavaScript\*. arXiv. https://doi.org/10.48550/arXiv.1212.2341

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

- Park, D., Ștefănescu, A., & Roșu, G. (2015). KJS: A complete formal semantics of JavaScript. In \*\*Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’15)\*\* (pp. 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, 67\*\*(5), 86–95. https://doi.org/10.1145/3624723

- Walli, S. R. (1995). The POSIX family of standards. \*\*StandardView, 3\*\*(1), 11–17. https://doi.org/10.1145/210308.210315