**Seminarul 3 — Partea 1 (Teorie, extins)**

# Seminarul 3 — Partea 1: Teorie (Obiecte și OOP în JS — prototipuri, excepții, memoization, async/await)

\*\*Scopul acestei părți:\*\* să înțelegi în profunzime cum funcționează modelul de obiecte al JavaScript (bazat pe \*prototype delegation\*), cum proiectezi ierarhii de erori utile domeniului tău, când și cum aplici \*memoization\* în siguranță și cum orchestrezi asincronia modernă cu \*\*async/await\*\* (incluzând \*timeouts\*, \*retry cu exponential backoff\*, \*cancellation\* cu `AbortController`). Întregul demers este ancorat într‑un scenariu realist (hook) și presărat cu exemple concrete, comparații din „lumea reală” și recomandări pentru folosirea asistată de AI (Copilot/ChatGPT) în regim \*\*VSL\*\* (Verify → Specify → Learn).

---

## Hook realist: „StudentHub” — când prototipurile și asincronia îți țin sistemul în viață

Imaginează‑ți că ești responsabil de platforma „\*\*StudentHub\*\*” a unei asociații studențești. Ai următoarele cerințe: (1) agregi date despre \*\*Studenți\*\*, \*\*Cluburi\*\* și \*\*Evenimente\*\* din surse interne (fișiere CSV/JSON) și dintr‑un endpoint HTTP; (2) validezi și normalizezi datele (nume, email, interese), (3) calculezi rapid \*top‑interests\* (popularitatea intereselor), (4) livrezi rapoarte în CLI sau în UI fără blocaje, iar (5) sistemul trebuie să‑și păstreze \*\*robustețea\*\* atunci când sursele sunt lente sau „capricioase”.

Într‑un campus real, asta seamănă cu un \*\*birou de evidență\*\*: studenții sunt dosare (obiecte), cluburile sunt departamente, iar evenimentele sunt fișe de planificare. Ai reguli (invariante), excepții (dosare incomplete, surse indisponibile), un registru cu răspunsuri rapide (cache/memoization) și o procedură pentru a nu ține oamenii la coadă la nesfârșit (timeouts, retry, limitare de concurență). În JavaScript, aceste nevoi se materializează în: \*\*obiecte\*\* cu prototipuri, \*\*ierarhii de erori\*\*, \*\*memoization\*\* și \*\*async/await\*\*.

În continuare, vom construi un fundament conceptual solid, pentru ca în Partea 2 (Laborator) să putem implementa „StudentHub Core” incremental, testabil și extensibil.

---

## 1. De ce OOP în JavaScript (și de ce „prototype delegation”, nu numai clase)?

JavaScript este \*\*multi‑paradigm\*\*: permite stil \*\*imperativ\*\*, \*\*funcțional\*\* (higher‑order, \*first‑class functions\*, imutabilitate) și \*\*obiectual\*\*. Spre deosebire de limbaje \*class‑based\* (Java, C#), modelul intrinsec JS este \*\*prototype‑based\*\*: fiecare obiect poate delega către un alt obiect (prototip) prin lanțul `[[Prototype]]`. Clasele (`class`) introduse în ES2015 sunt în esență \*\*syntactic sugar\*\* peste \*\*funcții constructor\*\* și \*\*delegation\*\*.

Avantaje practice:

- \*\*Reutilizare prin delegare\*\*: în loc să „copiem” comportamente între multe obiecte, le \*delegăm\* unui prototip comun.

- \*\*Flexibilitate\*\*: putem crea obiecte rapid cu `Object.create(proto)` și doar suprascriem ce e necesar.

- \*\*Compoziție > moștenire\*\*: evităm ierarhii adânci, preferând compoziția de capabilități (\*mixins\*).

Puncte de atenție:

- \*\*`this` binding\*\*: valoarea lui `this` depinde de modul de apel; \*arrow functions\* nu au `this` propriu.

- \*\*Moștenire multiplă\*\*: nu există nativ; folosim \*mixins\* sau compoziție funcțională.

- \*\*Iluzia claselor\*\*: `class` simplifică sintaxa, dar „motorul” rămâne `[[Prototype]]`. Înțelegerea delegării te ajută să explici comportamentul real la runtime.

\*\*Analogie:\*\* prototipul este „\*\*manualul de funcționare\*\*” ținut într‑un singur exemplar la bibliotecă. Când un student (obiect) nu știe să răspundă la o întrebare (nu are o metodă), merge la bibliotecă (prototip) să consulte manualul. Dacă manualul nu conține secțiunea, întreabă la bibliotecă un alt manual (prototipul prototipului) — lanțul continuă până la capăt (`Object.prototype`).

---

## 2. Modelul de obiecte: identitate, valoare, proprietăți, prototipuri

### 2.1. Identitate vs. valoare

Două obiecte pot avea \*\*aceleași câmpuri\*\* dar \*\*identități diferite\*\* (`{id:1} !== {id:1}`). Compararea prin „valoare” este \*\*aplicațională\*\* (definiție proprie), nu nativă. În OOP, \*\*identitatea\*\* contează pentru mutații, cache și colecții (`Map`, `Set`).

### 2.2. Proprietăți și descriptori

Fiecare proprietate are \*\*descriptor\*\*: `value`, `writable`, `enumerable`, `configurable`, sau `get/set`. Cu `Object.defineProperty` putem crea API‑uri precise (de ex. ascundem câmpuri interne prin `enumerable:false`).

### 2.3. Crearea obiectelor și prototipuri

- \*\*Obiect literal\*\*: rapid și expresiv.

- \*\*Funcție constructor\*\* + `new`: vechiul idiom.

- \*\*`class`\*\*: sintaxă modernă, cu `constructor`, `extends`, `super`.

- \*\*`Object.create(proto)`\*\*: creăm un obiect cu prototip explicit, fără constructor.

\*\*Delegation\*\*: la accesul unei proprietăți, motorul caută mai întâi pe obiectul curent; dacă nu o găsește, urcă pe `[[Prototype]]` ș.a.m.d.

### 2.4. `this`, `new`, `super`, `instanceof`

- `this` la invocare: `obj.m()` → `this === obj`. Cu \*arrow\*, `this` se leagă lexical.

- `new` creează automat obiectul, setează prototipul și apelează constructorul.

- `super` se referă la prototipul „părinte” (în sintaxa `class`).

- `instanceof` verifică lanțul `[[Prototype]]`; poate fi personalizat cu `Symbol.hasInstance`.

### 2.5. Simboluri utile și protocoale

- `Symbol.toStringTag` — \*branding\* (`Object.prototype.toString.call(x)` devine `"[object Student]"`).

- `Symbol.iterator` / `Symbol.asyncIterator` — \*iterator protocol\* (colectii iterabile).

- `Symbol.toPrimitive` — conversie custom în primitiv (`+obj`, `String(obj)`).

\*\*Comparație din lumea reală:\*\* `toStringTag` e ca o \*\*ștampilă\*\* pe dosar, arătând ce tip de dosar este. `iterator`/`asyncIterator` sunt ca \*\*dosare cu șină\*\*: îți oferă un mecanism standard de a răsfoi fișele, fie imediat (iterator), fie pe măsură ce vin din arhivă (async iterator).

---

## 3. Compoziție, mixins și \*factory functions\* (vs. moștenire adâncă)

### 3.1. De ce compoziție?

\*\*Compoziția\*\* asamblează „capabilități” într‑un obiect: de pildă, `Taggable(obj)` adaugă `addTag/removeTag/listTags`. Evităm \*diamantul morții\* (moștenire multiplă) și facem API‑uri \*\*clare și locale\*\*.

### 3.2. \*Mixins\*

Un \*mixin\* este o funcție care primește un obiect (sau un constructor) și îi adaugă metode. Important: documentează \*\*invariantele\*\* pe care le presupune (ex. existența unui `id`).

### 3.3. \*Factory functions\*

În loc de `class`, putem avea:

function makeStudent({ id, name, email, interests = [] }) {  
 if (!id || !name || !email) throw new ValidationError('Student fields required');  
 const self = Object.freeze({ id: String(id), name: String(name).trim(), email: String(email).toLowerCase().trim(), interests: interests.slice() });  
 return Object.setPrototypeOf(self, studentProto);  
}

Avantaje: claritate în inițializare, ușurință la testare, lipsa surprizelor cu `this`. Dezavantaje: lipsa sugar‑ului `class`, posibile costuri de memorie dacă nu delegăm la prototip.

### 3.4. LSP (Liskov Substitution Principle) în practică

Chiar dacă JS nu impune tipuri statice, \*\*contractele\*\* contează. Dacă expui o subclasă/variație, nu‑i rupe așteptările: metodele trebuie să păstreze semnătura și semantica de bază. În laborator vom prefera \*\*compoziția\*\* în fața moștenirii „adânci”.

---

## 4. Erori și excepții: cum proiectăm o ierarhie utilă domeniului („error taxonomy”)

### 4.1. `try/catch/finally` și `Error`

JS oferă `Error` și derivatele `TypeError`, `RangeError`, `SyntaxError` ș.a. Dar, pentru un domeniu precum „StudentHub”, e util să avem:

- `AppError` (baza comună), cu păstrarea `name` și `cause`.

- `ValidationError` (date incoerente), `NotFoundError` (căutări eșuate), `RemoteError` (eșec la sursa externă), `TimeoutError` (limite de timp).

### 4.2. Mesaje prietenoase, `cause` și \*logging\*

Un mesaj de eroare ar trebui să spună: \*\*ce\*\* s‑a întâmplat, \*\*de ce\*\* crezi că s‑a întâmplat, \*\*ce poate face utilizatorul\*\*. `cause` păstrează eroarea originală (context). În \*logging\*, evită date sensibile; atașează \*tags\* pentru filtrare.

### 4.3. Erori asincrone

În `async/await`, erorile sunt capturate ca reject ale promisiunii. Ai grijă la `try/catch` plasat corect. Pentru \*fan‑out/fan‑in\*, `Promise.all` oprește la primul reject; `Promise.allSettled` colectează toate rezultatele (bun pentru rapoarte parțiale).

### 4.4. \*Control flow\* prin excepții?

Evita să folosești excepțiile ca „if‑uri mai șmechere”. Excepțiile sunt pentru \*\*condiții excepționale\*\*, nu pentru ramuri obișnuite. În codul critic, preferă obiecte rezultat `Ok/Err` sau \*result types\* (pattern funcțional), dar menține o ierarhie clară pentru \*throw cases\*.

---

## 5. Memoization: când memoizăm, cum alegem chei stabile și cum evităm capcanele

### 5.1. Fundament (istoric & idee)

Termenul \*memo function\* (Michie, 1968) desemnează funcții care \*\*își amintesc\*\* rezultatele pentru intrări date. În practică, \*memoization\* este eficientă când:

- funcția este \*\*pură\*\* (același input → același output),

- costul de calcul e semnificativ,

- aceeași intrare reapare frecvent.

### 5.2. Chei de cache

Cheia trebuie să fie \*\*deterministă\*\*: pentru intrări identice logic, să fie aceeași. Variante:

- \*\*Serializare\*\* (`JSON.stringify(args)`), simplă dar sensibilă la ordine și `undefined`.

- \*\*`Map` multi‑param\*\*: folosești un \*trie\* de hărți (rar necesar).

- \*\*`WeakMap`\*\* pe \*\*obiect‑rădăcină\*\* (ex.: listele de studenți); evită scurgeri de memorie.

- \*\*Hashing stabil\*\* (funcție \*stable hash\*) pentru obiecte mari (cost de implementare).

### 5.3. Invalida­re și TTL

Un cache nu e „pentru totdeauna”. Definirea unei politici de invalidare (TTL, \*event‑based\*, \*manual\*) e obligatorie dacă datele se schimbă. Alternativ, aplicăm \*\*imutabilitate superficială\*\* la datele de intrare (freeze) și tratăm orice modificare ca \*\*nouă versiune\*\* (obiect nou).

### 5.4. Capcane

- \*\*Mutabilitate\*\*: dacă schimbi `students[0].interests` după ce ai salvat rezultatul, cache‑ul devine „stale”.

- \*\*Egalitate referențială\*\*: două array‑uri diferite dar cu același conținut vor genera chei diferite sub `JSON.stringify` (OK, dar interesează‑te dacă e acceptabil).

- \*\*Stat global\*\*: \*memoization\* pe funcții cu efecte colaterale poate ascunde bug‑uri; aplică numai la funcții pure sau izolează efectele.

---

## 6. Asincronie modernă: event loop, Promises, async/await, timeouts, retry, cancellation

### 6.1. \*Event loop\* pe scurt

Motorul JS rulează într‑un fir (single‑thread) și împarte lucrul în \*\*task queue\*\* (macrotasks: `setTimeout`, I/O) și \*\*microtask queue\*\* (`Promise` reactions). `await` introduce microtasks; ordinea lor e crucială în sincronizare.

### 6.2. Promises și `async`/`await`

`async function` returnează mereu o promisiune. `await` oprește execuția în acea funcție până când promisiunea se îndeplinește (fără a bloca firul). E ușor de compus:

const [a, b] = await Promise.all([getA(), getB()]); // fan-out/fan-in

`Promise.allSettled` colectează atât reușite, cât și eșecuri (bun pentru rapoarte parțiale).

### 6.3. Timeouts și retry/backoff

Niciun API nu e perfect. Împachetează apelurile incerte cu:

- \*\*`withTimeout(p, ms)`\*\* — respinge dacă depășește o limită, cu `TimeoutError`.

- \*\*`withRetry(fn, policy)`\*\* — reîncearcă funcția când eroarea este \*\*retryable\*\*, folosind \*\*exponential backoff\*\* (50ms, 100ms, 200ms…).

### 6.4. `AbortController`

Oferă posibilitatea de a \*\*anula\*\* operații în curs (acolo unde sunt \*abort‑aware\*). Transmiți `signal` în `fetch`/pipeline; dacă utilizatorul părăsește pagina sau UI anulează cererea, \*\*nu risipi resurse\*\*.

### 6.5. Limitarea concurenței: semafoare

Dacă ai un endpoint cu \*rate‑limit\*, rulezi cererile cu un \*\*semafor\*\* (`makeSemaphore(n)`), pentru a preveni avalanșa de cereri și erori în cascadă. În scop didactic, un semafor minimal ajunge; în producție, poți apela la cozi robuste.

\*\*Analogie din lumea reală:\*\*

- `withTimeout` = „nu stau la coadă la infinit”;

- `withRetry` = „încerc încă o dată, dar nu la nesfârșit și cu pauze din ce în ce mai mari”;

- `AbortController` = „am închis cererea, nu mai am nevoie de răspuns”;

- semafor = „intrarea pe rând la ghișeu”.

---

## 7. Securitate & robustețe: \*prototype pollution\*, validări, „fail‑fast”

### 7.1. \*Prototype pollution\*

Când importăm JSON din surse necontrolate, \*\*nu\*\* copia proprietăți precum `\_\_proto\_\_` sau `constructor`; folosește \*whitelisting\* explicit (extrage câmpurile permise). Evită `Object.assign(target, unchecked)` direct peste obiecte „vii”.

### 7.2. Validări sistematice

Ai „\*\*invariante\*\*” pentru fiecare entitate (Student: `id`, `name`, `email`). Verifică‑le \*\*la frontieră\*\* (constructor/factory). Pentru colecții mari, oferă \*\*rapoarte de validare\*\* (nu te opri la prima eroare când agregi date).

### 7.3. \*Fail‑fast\* și izolare

Când detectezi inconsistențe, „prinde‑le” \*\*devreme\*\* cu `ValidationError`. Izolează serviciile externe cu \*facade/adapters\*; dacă un endpoint e \*flaky\*, \*retry\* cu backoff și \*circuit breaker\* minimal (în L3).

---

## 8. Testabilitate: invariante ca teste, determinism, „oglindă” Vitest & Jest

### 8.1. Invariante codificate în teste

Dacă „obiectele sunt dosare”, atunci „\*\*regulile arhivei\*\*” sunt invariantele. Convertește‑le în teste: „Student trebuie să aibă `id`, `name`, `email`”; emailul se normalizează; obiectele sunt \*frozen\* (immutabile superficial).

### 8.2. Determinism și \*memoization\*

Testele trebuie să confirme că \*\*același input → același output\*\* și că \*memoization\* produce \*\*cache‑hit\*\* (aceeași referință) la repetare. Împerechează testele în \*\*Vitest & Jest\*\* pentru a evita „bias de framework”.

### 8.3. Teste asincrone

Folosește \*fake timers\* unde e rezonabil; verifică `withRetry` și `withTimeout`; simulează erori negative. Pentru orchestrări paralele, testează că nu depășești limita semaforului în nicio etapă.

---

## 9. AI‑assist (Copilot/ChatGPT/Mistral/Claude): metodologie VSL

- \*\*Verify\*\*: cere 10–15 \*edge‑cases\* (nume cu diacritice, spații, email invalid, interese `[]`, `null`, „aliasing” între referințe). Cere generarea de \*\*teste negative\*\*.

- \*\*Specify\*\*: impune \*\*contracte\*\* (semnături, clase de erori, mesaje exemplu). Cere păstrarea contractelor la refactor.

- \*\*Learn\*\*: cere \*refactor\* (de ex., `Map` → `WeakMap` + `TTL`) și compară \*trade‑off\*-urile.

- \*\*Guardrails\*\*: interzice modificarea semnăturilor publice; cere citarea \*edge‑cases\* în comentarii; verifică manual.

\*\*Prompts utile (exemple):\*\*

- „Propune 12 \*edge‑cases\* pentru `Student` și scrie teste Vitest/Jest care să demonstreze imutabilitatea superficială și normalizarea email‑ului.”

- „Oferă un `memoizeWeak(fn)` cu `WeakMap` pentru obiecte ca chei, explicând când nu e sigur să‑l folosesc.”

- „Rescrie `withRetry` cu \*exponential backoff\* și demonstrează tratamentul pentru erori \*non‑retryable\*.”

---

## 10. Anti‑pattern‑uri și checklist „gata de laborator”

### 10.1. Anti‑pattern‑uri

- Ierarhii adânci de moștenire pentru tipuri care nu respectă LSP.

- \*Memoization\* pe funcții cu efecte colaterale.

- Tratarea tuturor erorilor ca `Error` generic fără `name` sau `cause`.

- Lipsa limitării de concurență; apeluri paralele „nelimitate” la endpoint‑uri \*rate‑limited\*.

- `try/catch` care \*\*înghite\*\* eroarea fără \*rethrow\* sau fără \*logging util\*.

- Mutarea datelor de intrare după ce au fost folosite ca „chei de cache”.

### 10.2. Checklist „gata de laborator”

- [ ] Înțeleg lanțul de prototipuri și diferențele `class`/`Object.create`.

- [ ] Pot implementa `AppError` + derivate cu `cause`.

- [ ] Știu când e sigur să aplic \*memoization\* și cum generez chei stabile.

- [ ] Am un model mintal al `async/await`: `all`, `allSettled`, `race`, `any`.

- [ ] Pot explica `withTimeout`, `withRetry` (backoff), `AbortController`, `semafor`.

- [ ] Pot transforma invariantele în \*\*teste\*\* (Vitest & Jest).

---

## 11. Exemple scurte (doar pentru ancorare; codul complet va fi în Partea 2)

### 11.1. `Error` cu `cause`

class AppError extends Error {  
 constructor(message, { cause } = {}) {  
 super(message);  
 this.name = this.constructor.name;  
 if (cause) this.cause = cause;  
 }  
}  
class TimeoutError extends AppError {}

\*\*De ce:\*\* urmărire mai bună a erorilor, \*chain‑of‑cause\* pentru diagnostic.

### 11.2. `memoize` cu `Map`

const memoize = (fn) => {  
 const cache = new Map();  
 return (...args) => {  
 const key = JSON.stringify(args);  
 if (cache.has(key)) return cache.get(key);  
 const out = fn(...args);  
 cache.set(key, out);  
 return out;  
 };  
};

\*\*De ce:\*\* economie de timp la calcule deterministe costisitoare; atenție la chei stabile & imutabilitate.

### 11.3. `withTimeout`

const withTimeout = (promise, ms) => new Promise((resolve, reject) => {  
 const t = setTimeout(() => reject(new TimeoutError(`Timeout after ${ms}ms`)), ms);  
 promise.then(v => { clearTimeout(t); resolve(v); }, e => { clearTimeout(t); reject(e); });  
});

\*\*De ce:\*\* evită „cozile infinite”.

### 11.4. `AbortController` (schelet)

const controller = new AbortController();  
const p = fetch(url, { signal: controller.signal }); // sau un pipeline „abort-aware”  
controller.abort(); // ulterior, dacă e cazul

\*\*De ce:\*\* închide cererile nefolositoare, eliberând resurse.

---

## 12. Concluzie: o paradigmă pragmatică pentru sisteme robuste

\*\*Obiectele\*\* (cu prototipuri) oferă o cale naturală de a modela entități cu identitate și comportament delegat. \*\*Excepțiile\*\* și ierarhia de erori îți dau un vocabular controlat pentru eșecuri previzibile și imprevizibile. \*\*Memoization‑ul\*\* aduce performanță, dar cere \*\*determinism\*\* și \*\*disciplina datelor\*\*. \*\*Async/await\*\* simplifică asincronia, dar adevărata robustețe vine din \*\*timeouts\*\*, \*\*retry/backoff\*\*, \*\*cancellation\*\* și \*\*limitarea concurenței\*\*. În laboratorul din Partea 2, vei pune în mișcare toate aceste piese într‑un \*scaffold\* testabil, iar în Partea 3 vei exersa pe 45 de proiecte cu niveluri crescătoare de dificultate, inclusiv e2e și PWA pe subsetul avansat.

---

## Bibliografie (APA 7, cu DOI)

Maffeis, S., Mitchell, J. C., & Taly, A. (2008). An operational semantics for JavaScript. In \*Programming Languages and Systems\* (pp. 307–325). Springer. https://doi.org/10.1007/978-3-540-89330-1\_22

Michie, D. (1968). “Memo” functions and machine learning. \*Nature, 218\*(5136), 19–22. https://doi.org/10.1038/218019a0

Loring, M. C., Laurenzano, M. A., Newsham, Z., Hovsmith, N., Pande, S., Barik, T., & others. (2017). Semantics of asynchronous JavaScript. \*ACM SIGPLAN Notices, 52\*(11), 51–62. https://doi.org/10.1145/3170472.3133846

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

Ungar, D., & Smith, R. B. (1991). SELF: The power of simplicity. \*Higher‑Order and Symbolic Computation, 4\*(3), 171–216. https://doi.org/10.1007/BF01806105

Ungar, D., & Smith, R. B. (1987). Self: The power of simplicity. In \*OOPSLA ‘87\*. ACM. https://doi.org/10.1145/38765.38828