5. Advanced JavaScript Programming

Adrian Adiaconitei

LINK Academy

Obiective

- ✓ Recapitulare
  - ✓Tratarea erorilor
- ✓ Design patterns
- ✓ Hoisting
- ✓ Promise
- ✓ Async și await
- ✓ SOLID - OOP

JavaScript – Tratarea erorilor

JavaScript – Tratarea erorilor

- Erorile în aplicaţii pot fi împărţite în două tipuri:
  - **De sintaxă** (erori în sintaxă, se corectează uşor)
  - **Logice** (eroare logică, se corectează mai greu)
- De asemenea, erorile se pot împărţi în momentul evenimentului:
  - **Erori în compilare**
  - **Erori în executare (erori runtime)**

JavaScript – Tratarea erorilor

- ✓ Noi, în calitate de dezvoltatori, anticipăm şi să evităm aceste erori
- ✓ Java Script conţine un mecanism care permite ca aplicaţia să publice într-un mod „liniştit“ dacă ceva nu este în regulă, oferind dezvoltatorului posibilitatea de a reacţiona în momentul respectiv. Acest mecanism este sistemul de **erori (denumite și excepții în alte limbaje)**.
- ✓ **Tratarea erorilor** este situaţia în care o parte din program nu este în stare să funcţioneze conform unui plan prevăzut, dar, spre deosebire de terminarea programului, permite programului să execute o alternativă care va reabilita şi va evita întreruperea funcţionării programului

JavaScript – Tratarea erorilor

try un bloc de cod cd trebuie executat/rulat

catch un bloc de cod pentru a gestiona orice eroare.

finally un bloc de cod de rulat indiferent de rezultat.

throw definește o eroare personalizată.

JavaScript – Tratarea erorilor

- ✓ Cea mai frecventă situaţie de procesare a excepţiei este captarea (prinderea) sa.
- ✓ Pentru captarea excepţiilor se folosesc blocurile **try / catch.**
- ✓ Blocurile try catch funcţionează după principiul următor: Se creează structura:

try { *// codul care se execută și poate genera excepția* }
catch (error) { *// codul care se execută dacă apare excepția* }

- ✓ O parte din cod în care se aşteaptă excepţia se pune în blocul **try**
- ✓ O parte din cod care ar trebui să se execute dacă nu reuşeşte executarea primului bloc se pune în blocul **catch**

JavaScript – Tratarea erorilor

```
var y = 1;
try {
        allert("Hello!");
        y = 0;
}
catch (error) {
        console.log("I'm sorry, I couldn't find this method.");
}
console.log(y); // ? 1 sau 0
```

JavaScript – Tratarea erorilor

- ✓ Erorile se pot genera şi manual, în orice moment de executare a programului
- ✓ O eroare se poate arunca manual cu comanda **throw**.
- ✓ Comanda throw trebuie urmată de un obiect din clasa error **Error,** un obiect care moștenește clasa **Error** sau de un string:

throw 'something went terribly wrong’;

throw new Error('You cannot go outside today’);

throw new TypeError('Cannot convert a number to uppercase');

JavaScript – Tratarea erorilor

Tipul Erorii	Descriere
✓ **EvalError**	Eroare de evaluare
✓ **RangeError**	Un numar nu este in afara limitelor Referinta ilegala
✓ **ReferenceError**	
✓ **SyntaxError**	Eroare de sintaxa
✓ **TypeError**	Eroare de tip de date
✓ **URIError**	Eroare in decodarea unui url

JavaScript – Hoisting

În Java Script, o variabilă poate fi declarată după ce a fost folosită:

```
x = 5;

console.log(x);

var x;
```

JavaScript – Hoisting

Aceasta este posibil deoarece Java Script "ridică" toate declarațiile de variabile la începutul scopului curent înainte să execute codul:

Ap1.html

```
var x;

x = 5;

console.log(x);
```

JavaScript – Hoisting

Aceasta este adevărat doar pentru variabilele declarate cu var.
Variabilele și constantele declarate cu let sau const nu sunt "ridicate"!

Ap2.html

```
x = 5; //Uncaught ReferenceError: Cannot access 'x' before initialization

console.log(x);

let x;
```

JavaScript – Hoisting

Hoisted

- ✓ var
- ✓ function

Not Hoisted

- ✓ let
- ✓ const
- ✓ class

JavaScript – Design patterns

- ✓ Design patterns - soluții reutilizabile la problemele frecvente în proiectarea software.

- ✓ Design patterns - ne oferă, de asemenea, un vocabular comun pentru a descrie soluțiile.

JavaScript – Design patterns

- ✓ **Factory -** este un obiect care creează alte obiecte.
- ✓ Dacă aplicația dvs. are nevoie de mai mult control asupra procesului de creare a obiectelor, luați în considerare utilizarea unei fabrici.

JavaScript – Design patterns

- ✓ **Singleton** - limitează numărul de instanțe ale unui anumit obiect la doar unul.

JavaScript – Design patterns

- ✓ **Mediator/Middleware Pattern -**face posibil ca componentele să interacționeze între ele printr-un punct central: mediatorul.
- ✓ În loc să vorbească direct unul cu celălalt, mediatorul primește cererile și le trimite înainte!
- ✓ În JavaScript, mediatorul nu este adesea altceva decât un obiect literal sau o funcție.

Ap3.html

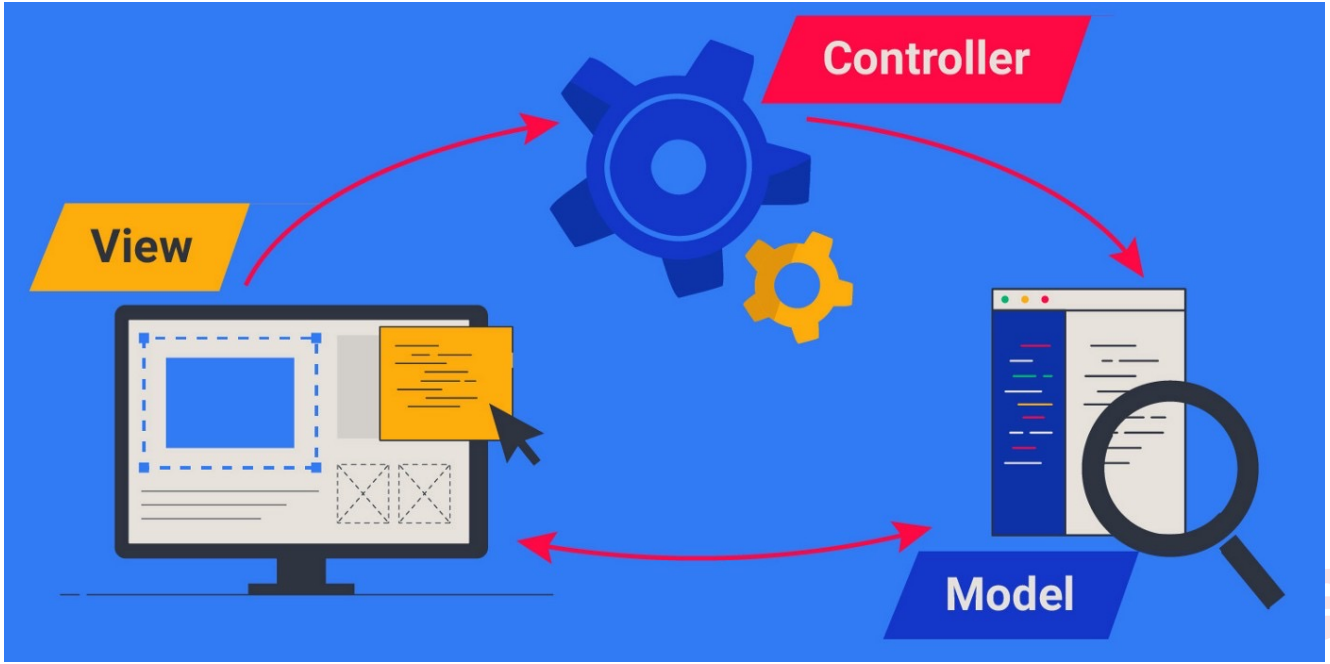
JavaScript – Design patterns

- ✓ **Observer Pattern-** putem abona anumite obiecte, observatorii, unui alt obiect, numit observabil. Ori de câte ori are loc un eveniment, observabilul își anunță toți observatorii!

- ✓ Un obiect observabil conține de obicei 3 părți importante:
- ✓ **subscribe**(): o metodă pentru a adăuga observatori la lista de observatori
- ✓ **unsubscribe**(): o metodă pentru a elimina observatorii din lista de observatori
- ✓ **notify**(): o metodă de a notifica toți observatorii ori de câte ori are loc un anumit eveniment
- ✓ **Ap4.html**

JavaScript – Design patterns

✓ **MVC Pattern-** Model, View și Controller.



Vue.js, Angular, Ember.js, Maria.js

JavaScript – Design patterns

✓**MVC Pattern-** Model, View și Controller.

Ap5.html

Hierarchical Model View Controller (**HMVC**)

Model View Adapter (**MVA**)

Model View Presenter (**MVP**)

Model View ViewModel (**MVVM**)

Ap6.html

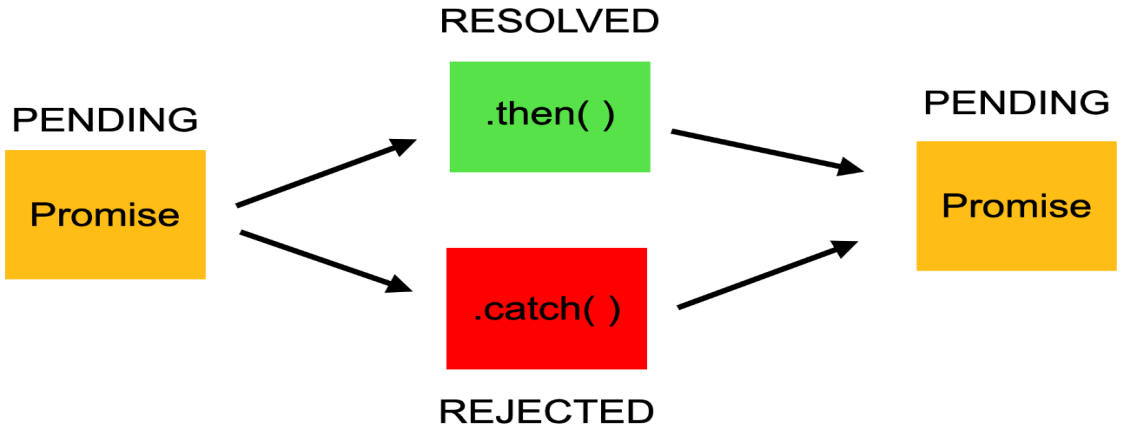
Promises

O promisiune: un obiect JavaScript care leagă un cod care poate dura ceva timp și un cod care trebuie să aștepte rezultatul ECMAScript 2015(ES6)

- **pending**: stare inițială, în asteptare, nici îndeplinită, nici respinsă
- **fulfille**: operațiunea a fost finalizată cu success
- **rejected**: operațiunea a eșuat

https://www.w3schools.com/js/js_promise.asp

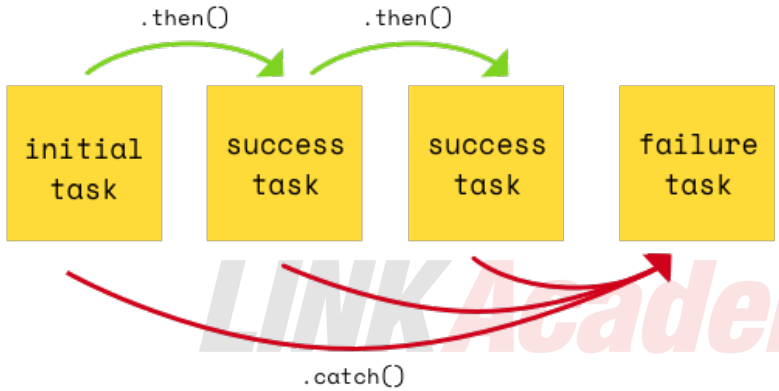
Ap7.html



Promises

Dacă este nevoie de a executa două sau mai multe operații asincrone, astfel încât fiecare operație ulterioară să înceapă atunci când operația anterioară reușește, de asemenea, operația ulterioară ia rezultatul operației anterioare. Acest lucru se realizează prin crearea **Promise Chaining** .

Ap7_1.html Ap7_2.html
Ap8.html



Async/await

```
async function f() {
return 1;
}
f().then(alert);
```

```
async function f() {
Return Promise.resolve(1);
}
 f().then(alert);
```

Async/await

Async/await – este **Syntactic sugar** pentru **PROMISES**

Syntactic sugar

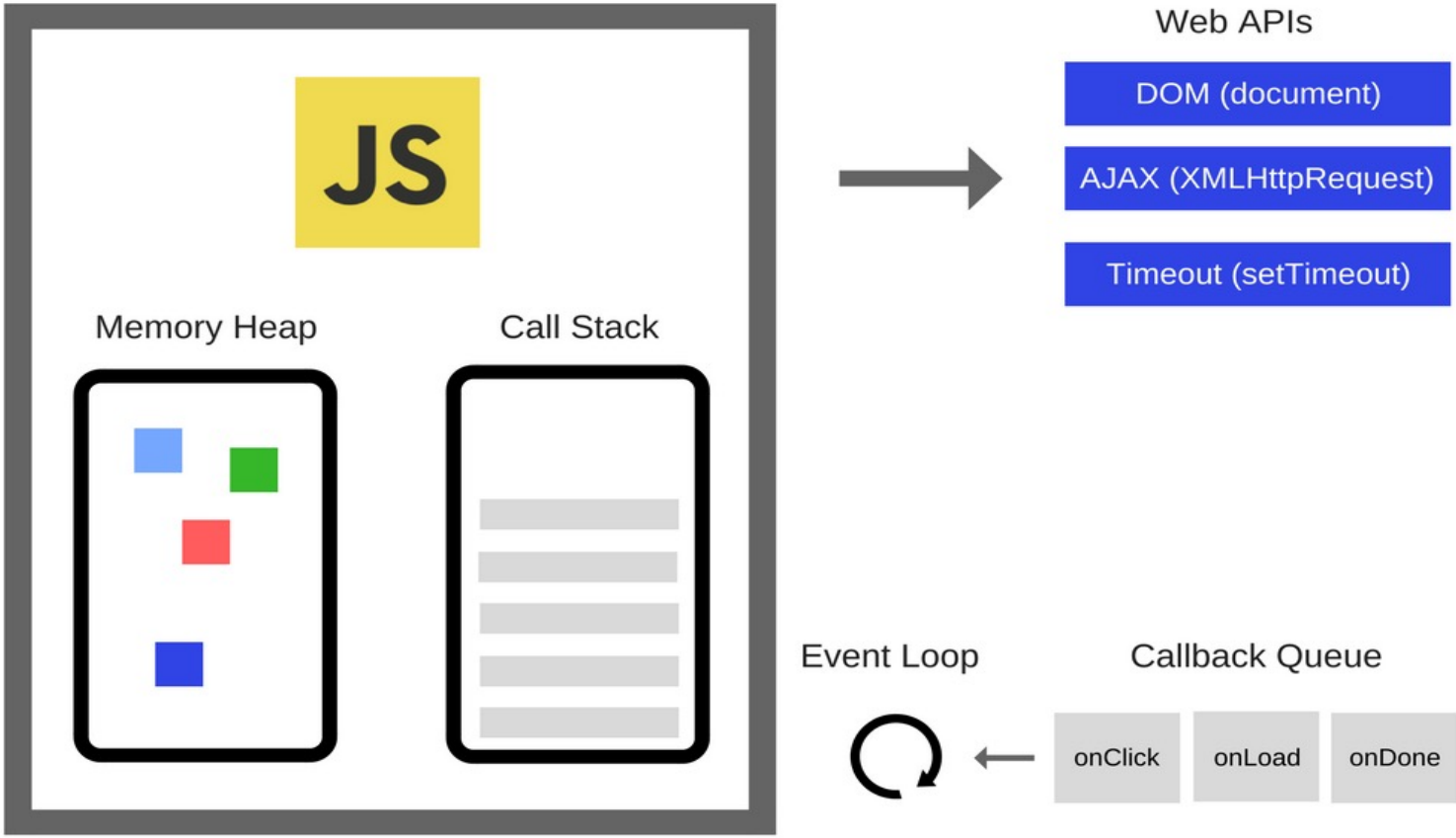
- ✓ funcționalitate proiectată pentru rescrierea altei funcționalități făcând codul mai ușor de înțeles
- ✓ NU ne permite să facem ceva ce nu puteam face și înainte folosind alte metode(PROMISES)

Async/await

- ✓ Există două moduri principale de a gestiona codul asincron: then/catch (ES6 - 2015) și **async/await** (ES8-2017)
- ✓ Marele avantaj al async / await este că face ca codul asincron să pară sincron. (fă cererea, așteaptă să se termine și apoi dă-mi rezultatul)
- ✓ **Async** face ca o funcție să returneze o Promisiune
- ✓ **Await** face ca o funcție să aștepte pentru o Promisiune
- ✓ **Await** poate fi utilizat numai în cadrul unei funcții **async**.
- ✓ **Await** oprește execuția unei funcții asincrone până când Promisiunea este rezolvată. Ap9.html

Async/await

Ap10.html

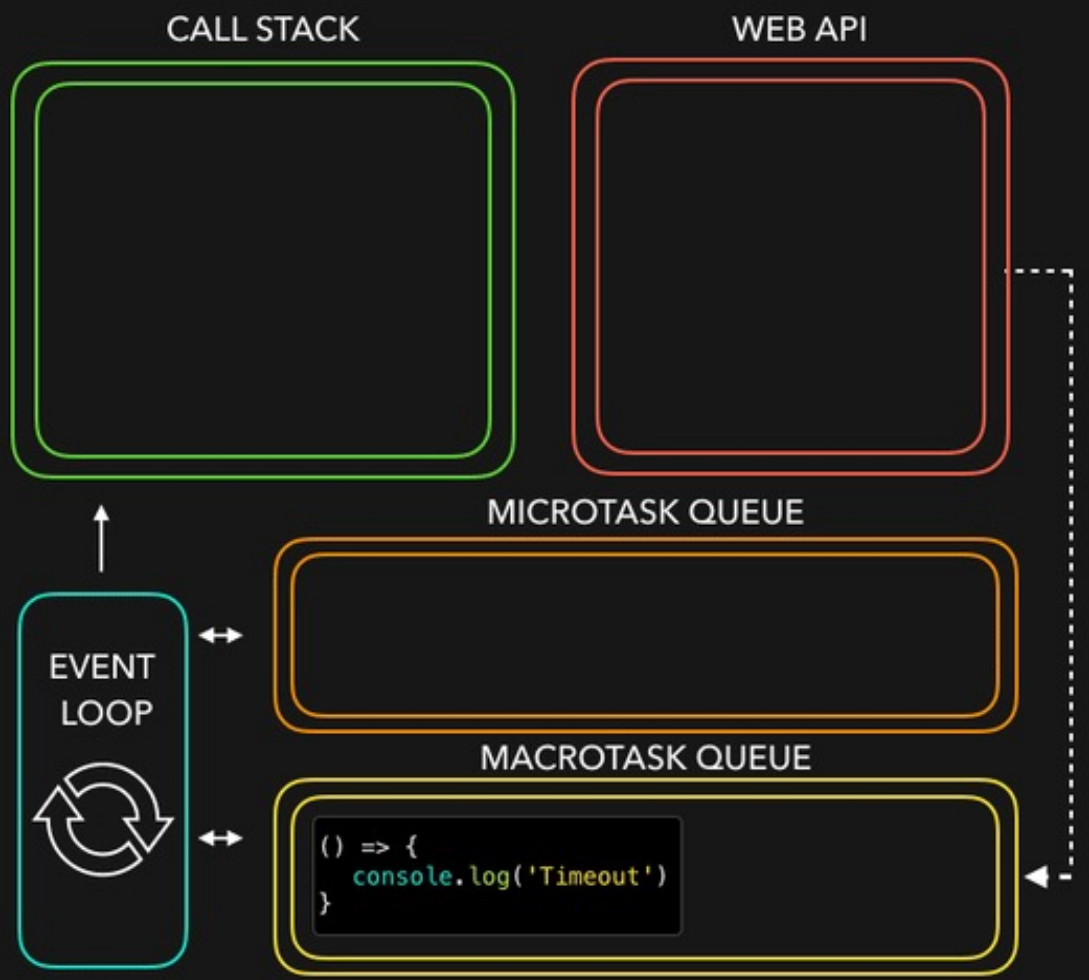


LINKAcademy

```
console.log('Start!')

setTimeout(() => {
  console.log('Timeout!')
}, 0)

Promise.resolve('Promise!')
   .then(res => console.log(res))

console.log('End!')
```

node

```
Start!
End!
Promise!
```



https://www.jsv9000.app/

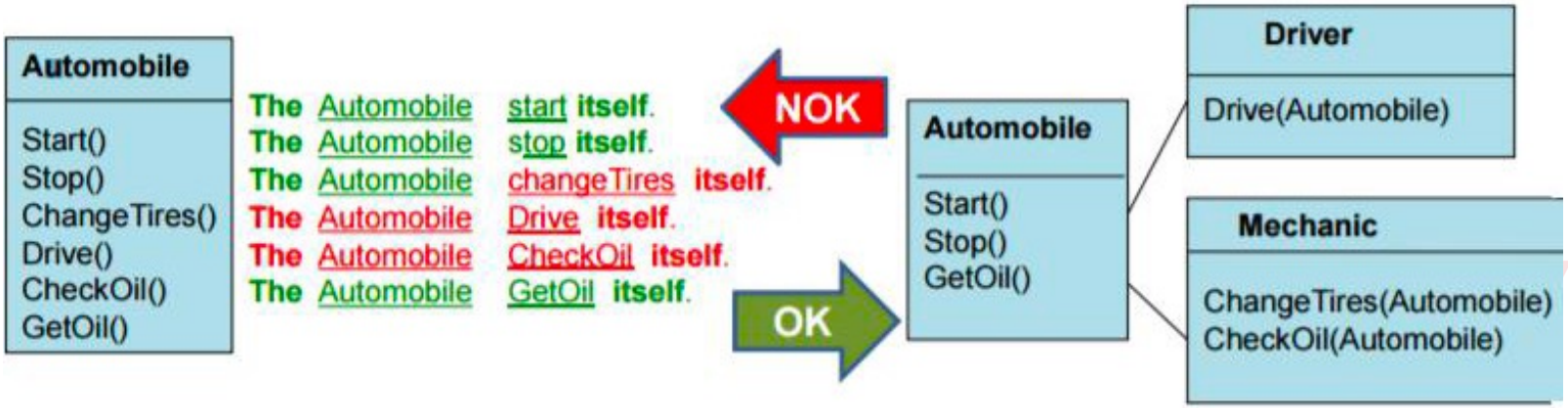
JavaScript – OPP : SOLID

- ✓ **S** — Single responsibility principle
- ✓ **O** — Open closed principle
- ✓ **L** — Liskov substitution principle
- ✓ **I** — Interface segregation principle
- ✓ **D** — Dependency Inversion principle

- ✓ Aceste 5 principii vă vor ghida cum să scrieți un cod mai bun.
- ✓ Sunt concepute pentru a ajuta dezvoltatorii să proiecteze aplicații robuste și care pot fi ușor întreținute.
- ✓ Principiile SOLID au fost introduse pentru prima dată de Robert J. Martin (alias Uncle Bob)

JavaScript – OPP : SOLID

- ✓ **Single responsibility principle**: orice clasă/funcție ar trebui să facă un singur lucru, prin urmare, ar trebui să aibă un singur motiv pentru a se schimba. O clasă sau funcție, nu ar trebui să rezolve sau să trateze mai mult decât un singur scop deoarece aceasta ar introduce cuplarea între cele două funcționalități.
- ✓ Soluția: formând propoziţii care se termină cu cuvântul **itself**



JavaScript – OPP : SOLID

- ✓ **Open closed principle**: orice clasă trebuie să poată fi extinsă oricând este nevoie, dar nu trebuie modificată. Permite schimbări fără a modifica codul existent. Folosirea moştenirii pentru a extinde/schimbacodulfuncţional existent şi a nu se atinge de codul care funcţionează (**public** și **private**)

- ✓ **Liskov substitution principle:** orice clasă derivată poate înlocui clasa de bază și codul să funcționeze corect în continuare. Altfel spus noile clase extinse din părinte trebuie să fie capabile să înlocuiască clasa de bază în toate funcțiile sale fără a fi nevoie să aducem modificări.

JavaScript – OPP : SOLID

- ✓ **Interface Segregation Principle** : în loc să avem o interfață cu foarte multe metode, din care doar câteva sunt folosite, mai bine avem mai multe interfețe mai mici și le folosim doar pe cele necesare. Un client nu ar trebui să fie obligat să implementeze o interfață pe care nu o folosește sau un client nu ar trebui să depindă de metode pe care nu le folosesc.

- ✓ **Dependency Inversion Principle:** dacă o clasă are o dependență, această dependență ar trebui să fie o interfață, nu o clasă concretă – acest principiu încurajează decuplarea claselor și e necesar pentru testarea unitară

Resurse

- ✓ https://ui-patterns.com/patterns
- ✓ https://www.dofactory.com/javascript/design-patterns
- ✓ https://www.patterns.dev/posts/classic-design-patterns/
- ✓ https://blog.sessionstack.com/how-javascript-works-writing-modular-and-reusable-code-with-mvc-16c65cbd9f64