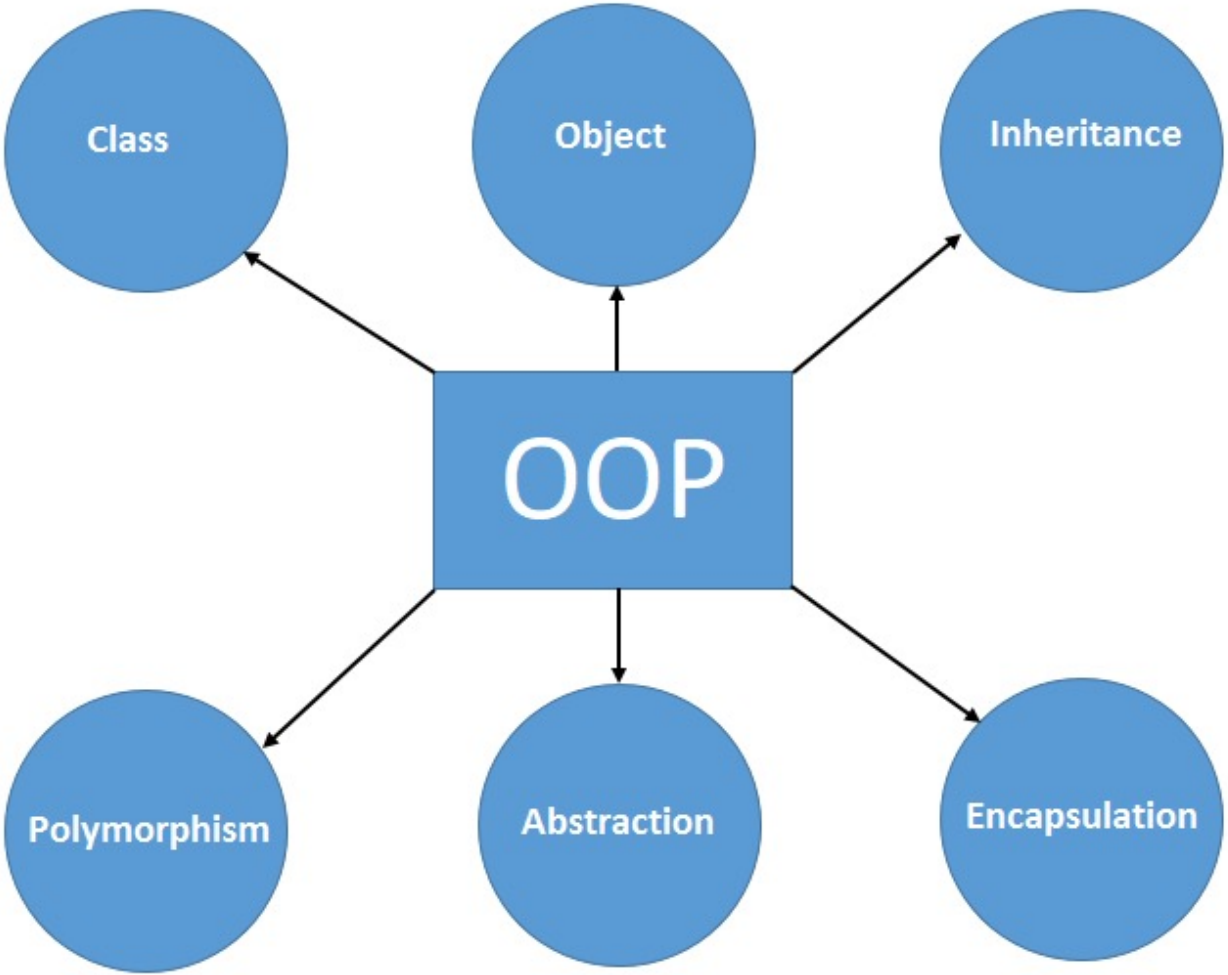
3. Advanced JavaScript Programming

Adrian Adiaconitei

LINK Academy

Obiective

- ✓ Recapitulare
- ✓ OOP în JavaScript
  - ✓ Abstractizare
  - ✓ Polimorfism

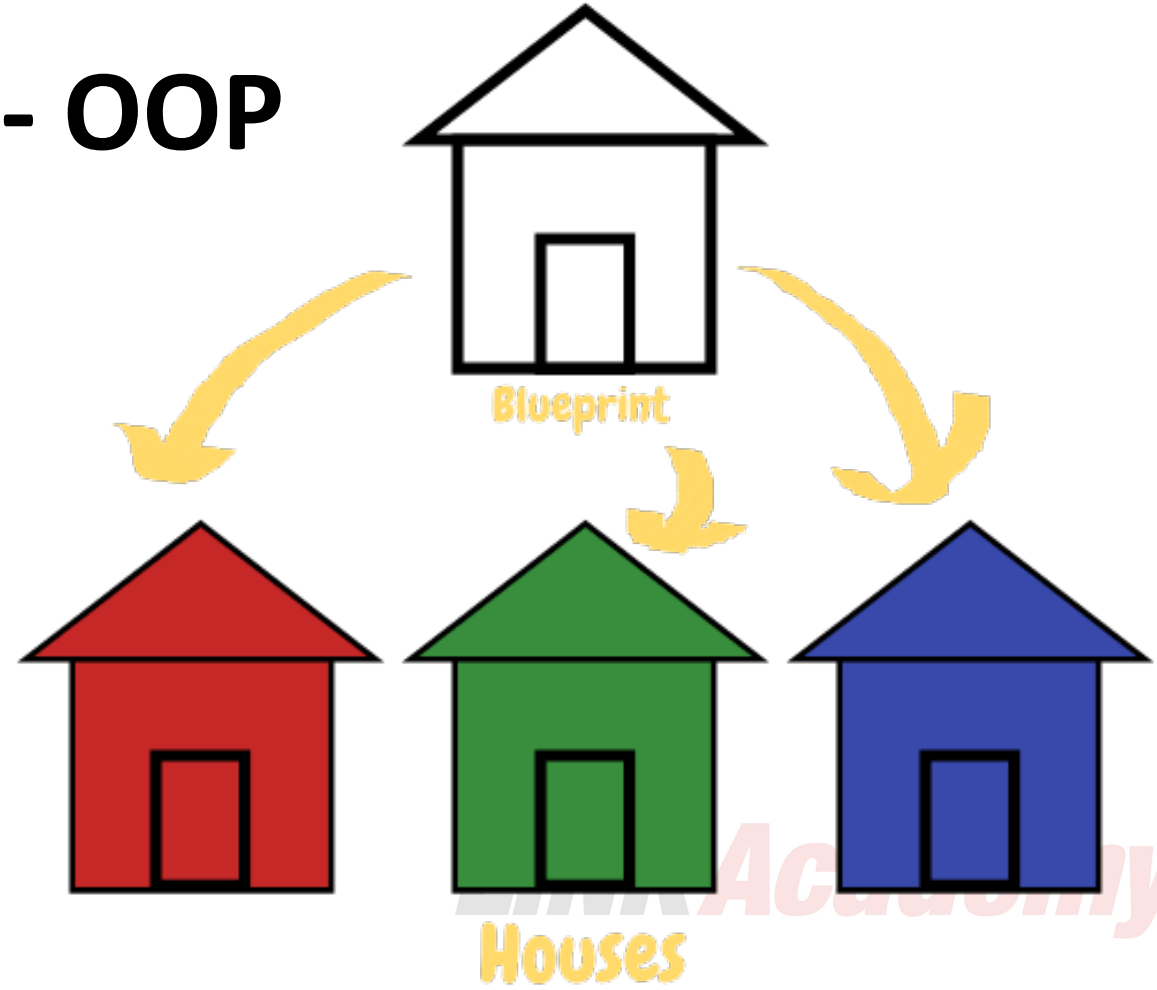


JavaScript - OOP

Beneficiile Programarii Orientate pe Obiecte

1. **Modularitatea**: codul sursă pentru o clasă poate fi scris și menținut independent de codul sursa pentru alte clase. Odată creat, un obiect poate fi ușor transferat în interiorul sistemului.
2. **Securitate** ridicată la nivel de cod: Interacționand doar cu metodele obiectului, detaliile implementării interne ramân ascunse de lumea exterioara.
3. **Reutilizarea codului**: dacă o clasă exista deja, puteți utiliza în programul dvs. obiecte din acea clasă. Acest lucru permite programatorilor să implementeze / testa / depana obiecte complexe.
4. **Debugging** mai usor: Dacă un anumit obiect se dovedește a fi o problema, il puteți elimina din program și puteti adăuga/conecta, ca inlocuitor, un obiect diferit.

JavaScript - OOP



JavaScript – OOP - Module

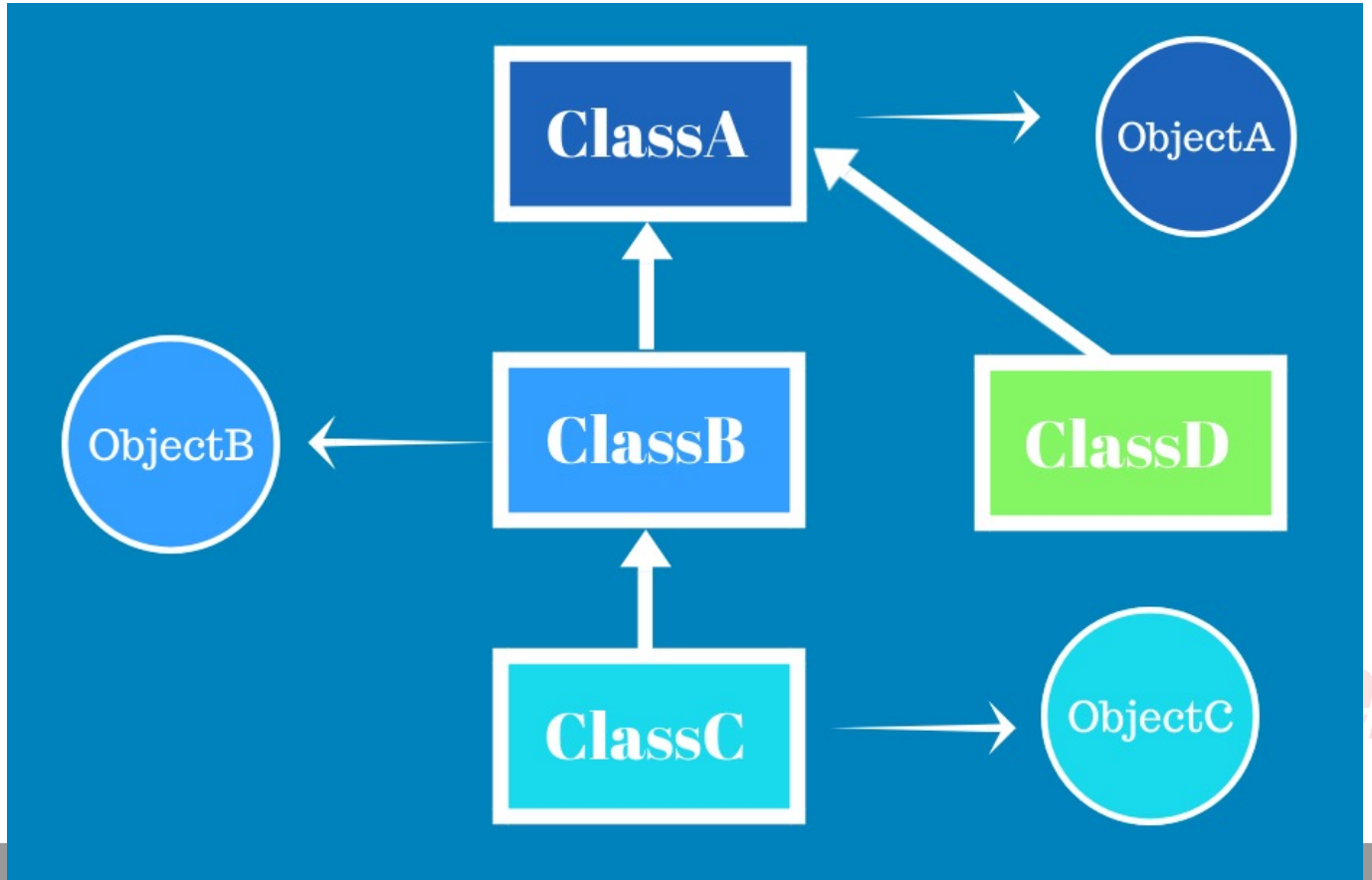
„ Autorii buni își împart cărțile în capitole și secțiuni;
programatorii buni își împart programele în module.”

- ✓ **Mentenabilitate:** un modul este autonom. Actualizarea unui singur modul este mult mai ușoară atunci când modulul este decuplat de alte bucăți de cod.
- ✓ **Spațierea numelor:** modulele pot rezolva conflictele de nume. **Namespace** este paradigma de programare de a oferi un domeniu de aplicare identificatorilor (nume de tipuri, funcții, variabile etc.) pentru a preveni coliziunile între ei.
- ✓ **Reutilizare:** importăm și folosim mai ușor codul când este modularizat

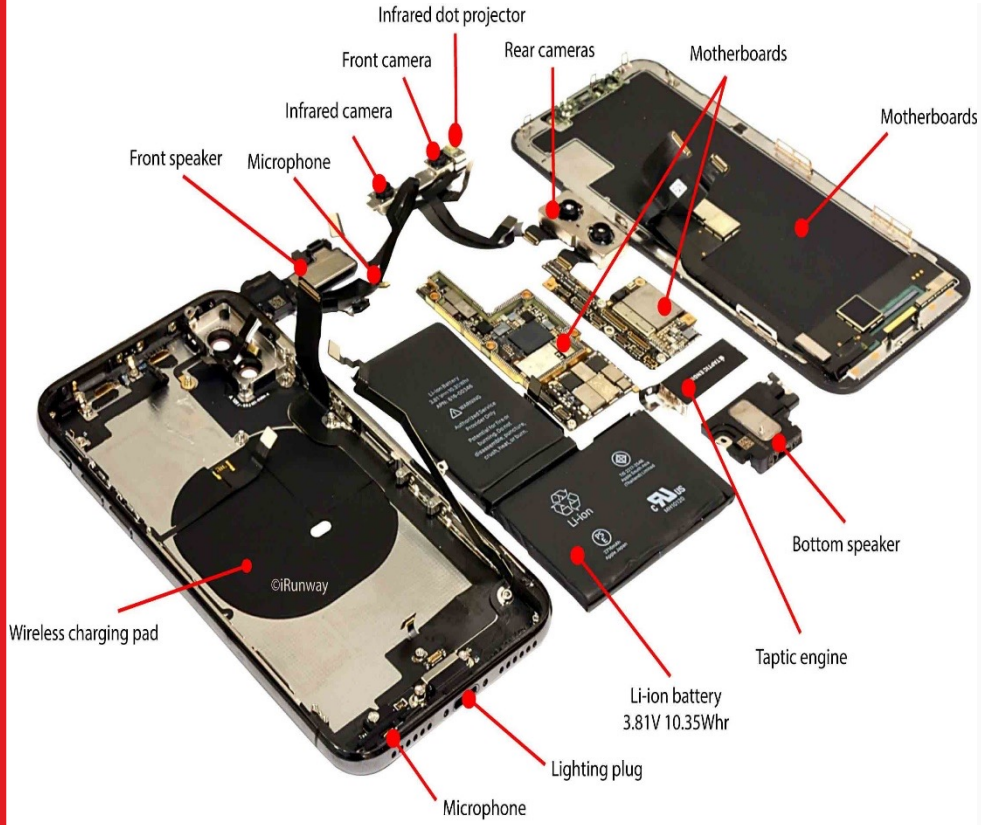
JavaScript – OOP- Moștenire

- ✓ cuvântul cheie **extends**, prin care acesta poate indica faptul ca o clasă este derivată dintr-una deja existenta.
- ✓ Clasa derivate(copil / subclasa) preia în acest fel, parțial sau total, atributele și metodele clasei originale(părinte / superclasa)
- ✓ cuântul cheie **super** din clasa copil apelează constructorul sau alta metodă din clasa părinte
- ✓ Deoarece **super**() inițializează **this**, trebuie să apelați **super**() înainte de a accesa **this**.
- ✓ Dacă în clasa copil nu este definit constructorul se apelează automat constructorul părinte

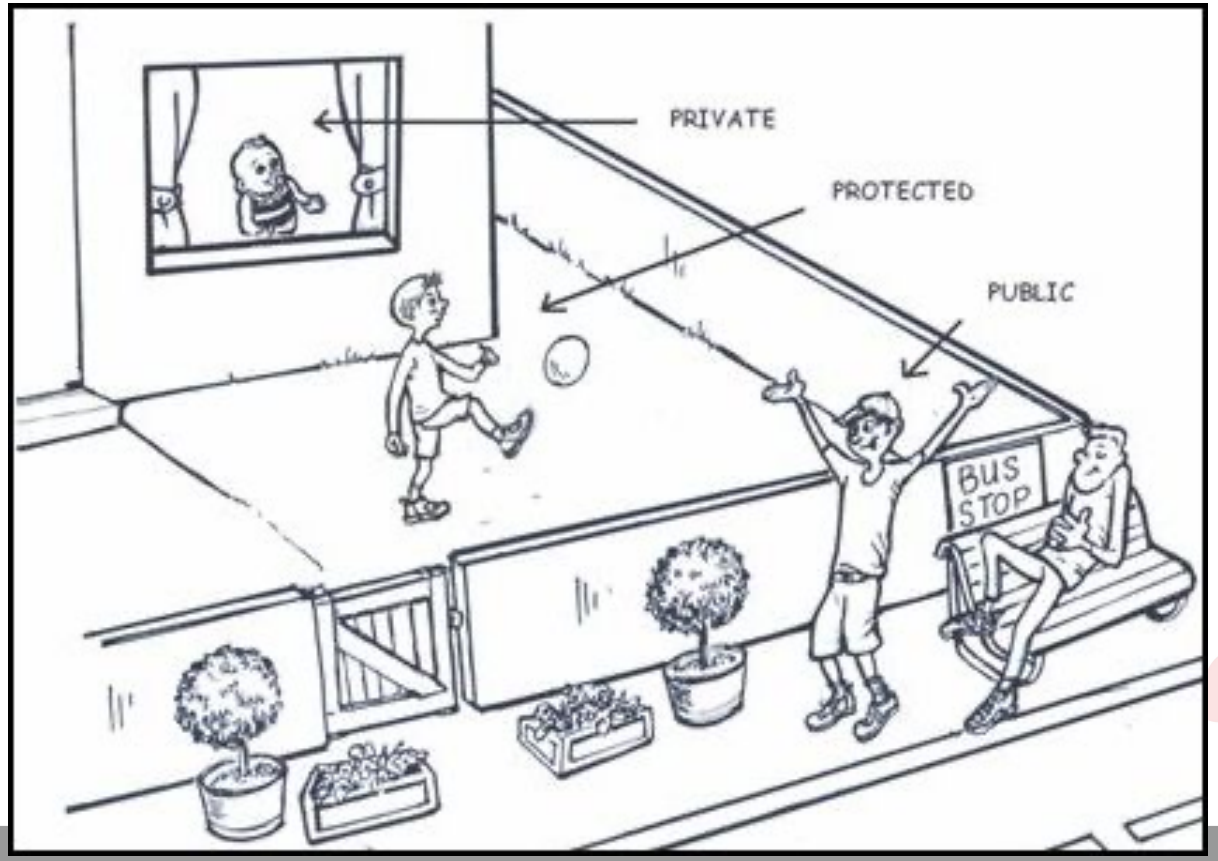
JavaScript – OOP- Moștenire



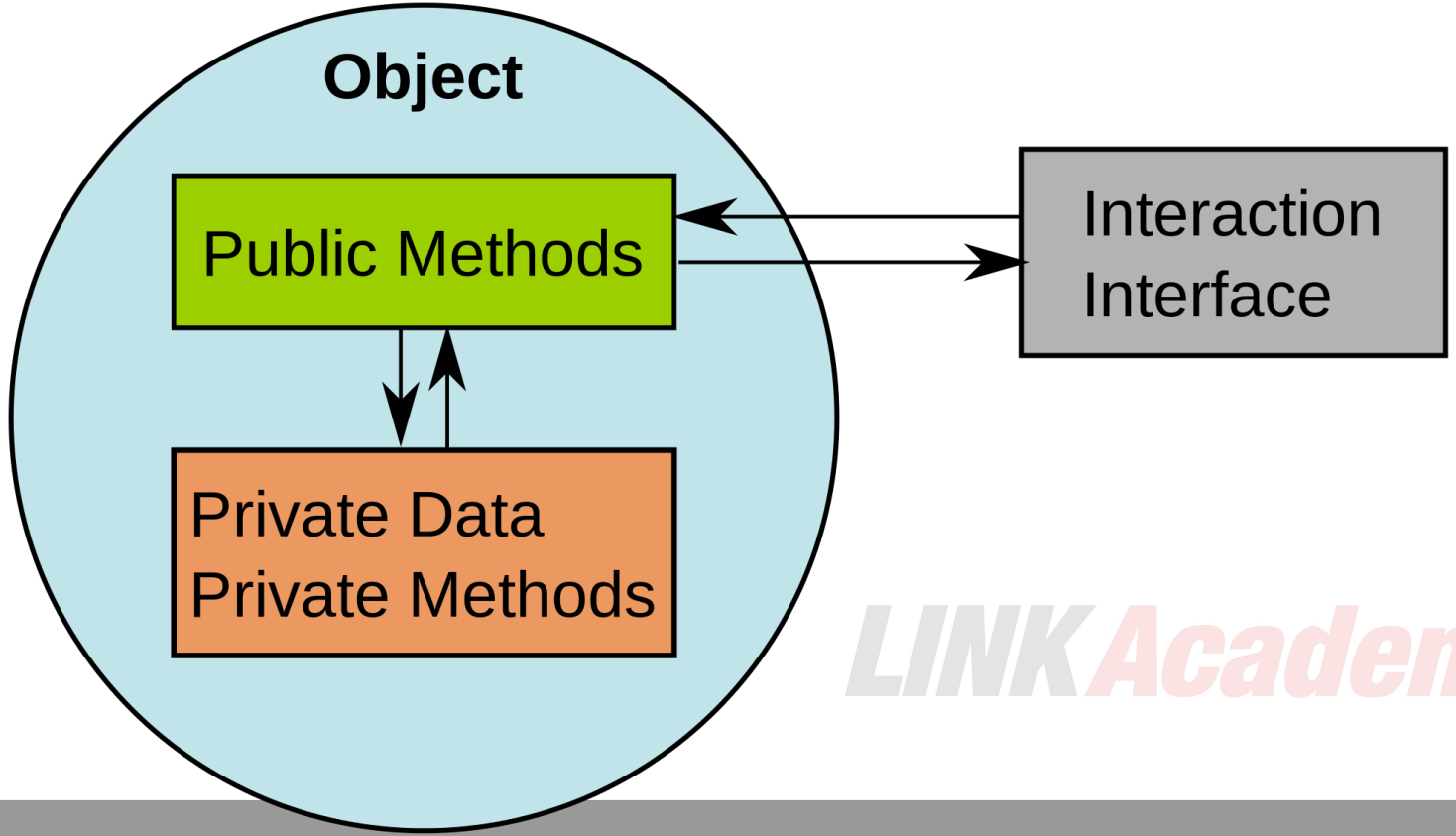
JavaScript – OOP– Încapsulare



JavaScript – OOP– Încapsulare



JavaScript – OOP– Încapsulare

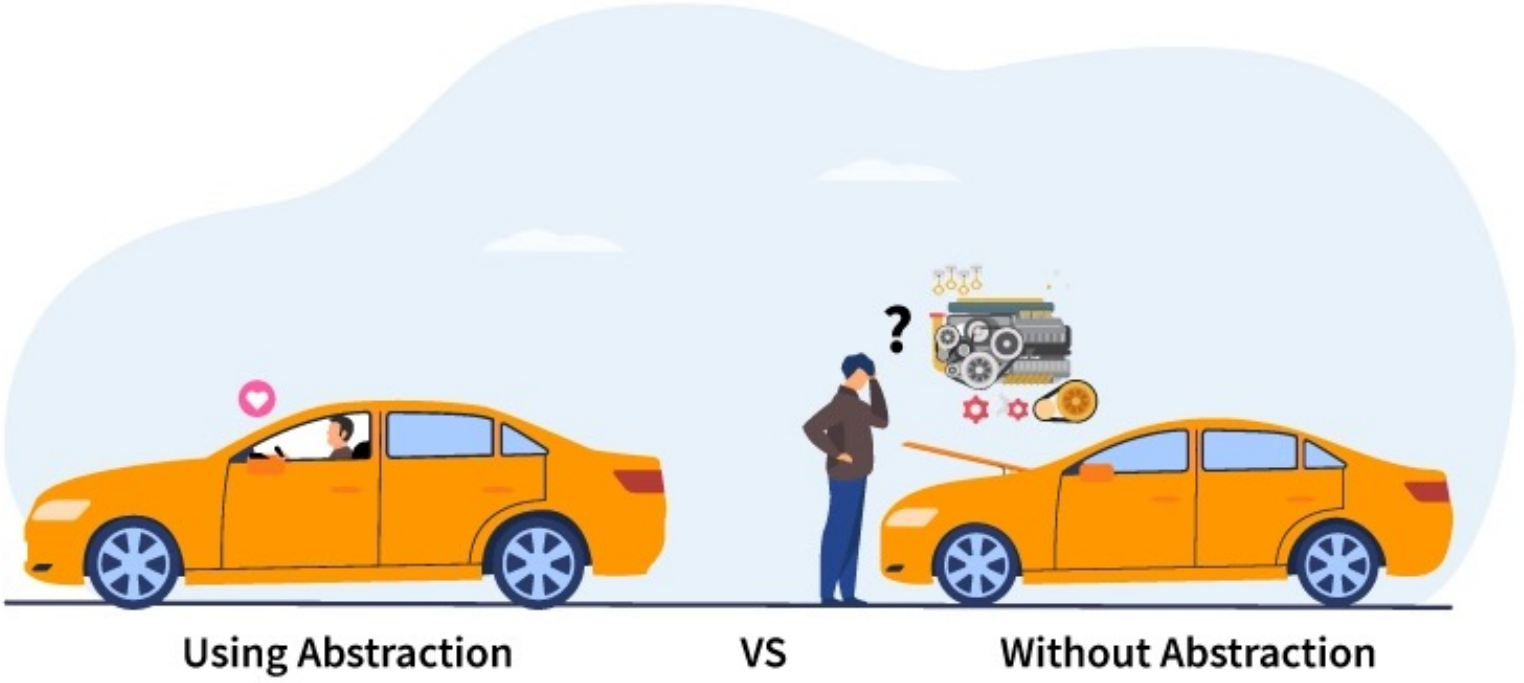


JavaScript – OOP– Încapsulare

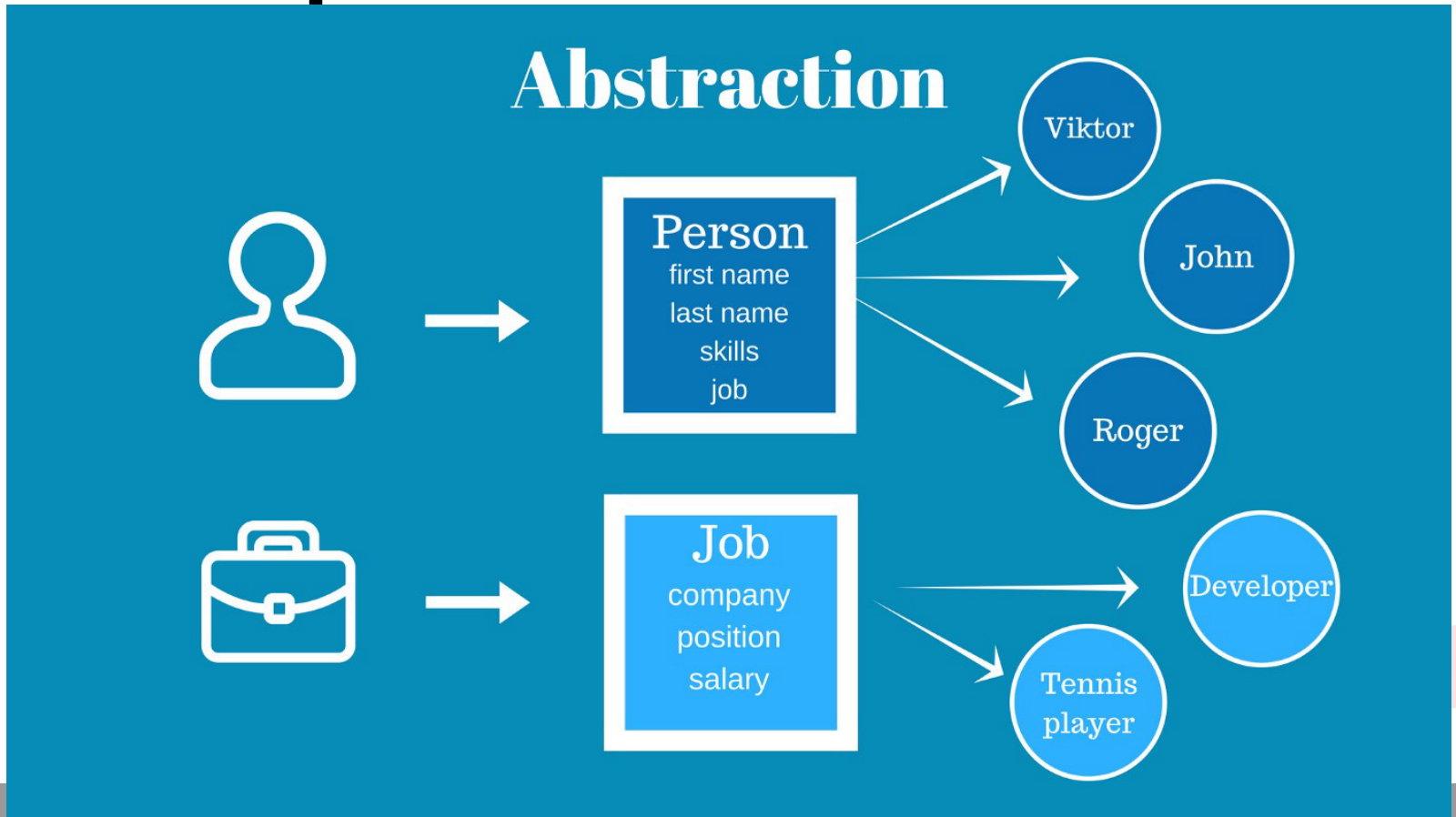
- Ap1.html – deosebirea dintre public și privat

LINKAcademy

JavaScript – OOP– Abstractizare



JavaScript – OOP– Abstractizare



JavaScript – OOP– Abstractizare

- ✓ Abstractizarea:este o modalitate de a reduce complexitatea și permite proiectarea și implementarea eficientă în sisteme software complexe.
- ✓ Ascunde complexitatea tehnică a sistemelor în spatele API-urilor. Ideea principală a abstractizării este definirea componentelor din viața reală în diferite tipuri de date complexe
- ✓ **Avantajele abstractizării**
  - Evită duplicarea codului și crește reutilizabilitatea.
  - Poate schimba implementarea internă a clasei în mod independent, fără a afecta utilizatorul.
  - Ajută la creșterea securității unei aplicații, deoarece numai detaliile importante sunt furnizate utilizatorului.

JavaScript – OOP– Abstractizare

- ✓ Abstractizarea înseamnă definirea unui schelet obligatoriu ce trebuie implementat (Interfețe)
- ✓ **Nu se poate crea o instanță a unei clase abstracte**
- ✓ **Metodele abstractă trebuie implementate.**

JavaScript – OOP– Abstractizare

Ap2.html

Class Person

Class Job

Ap3.html

Ap4.html

Ap5.html

JavaScript: Încapsulare – Abstractizare

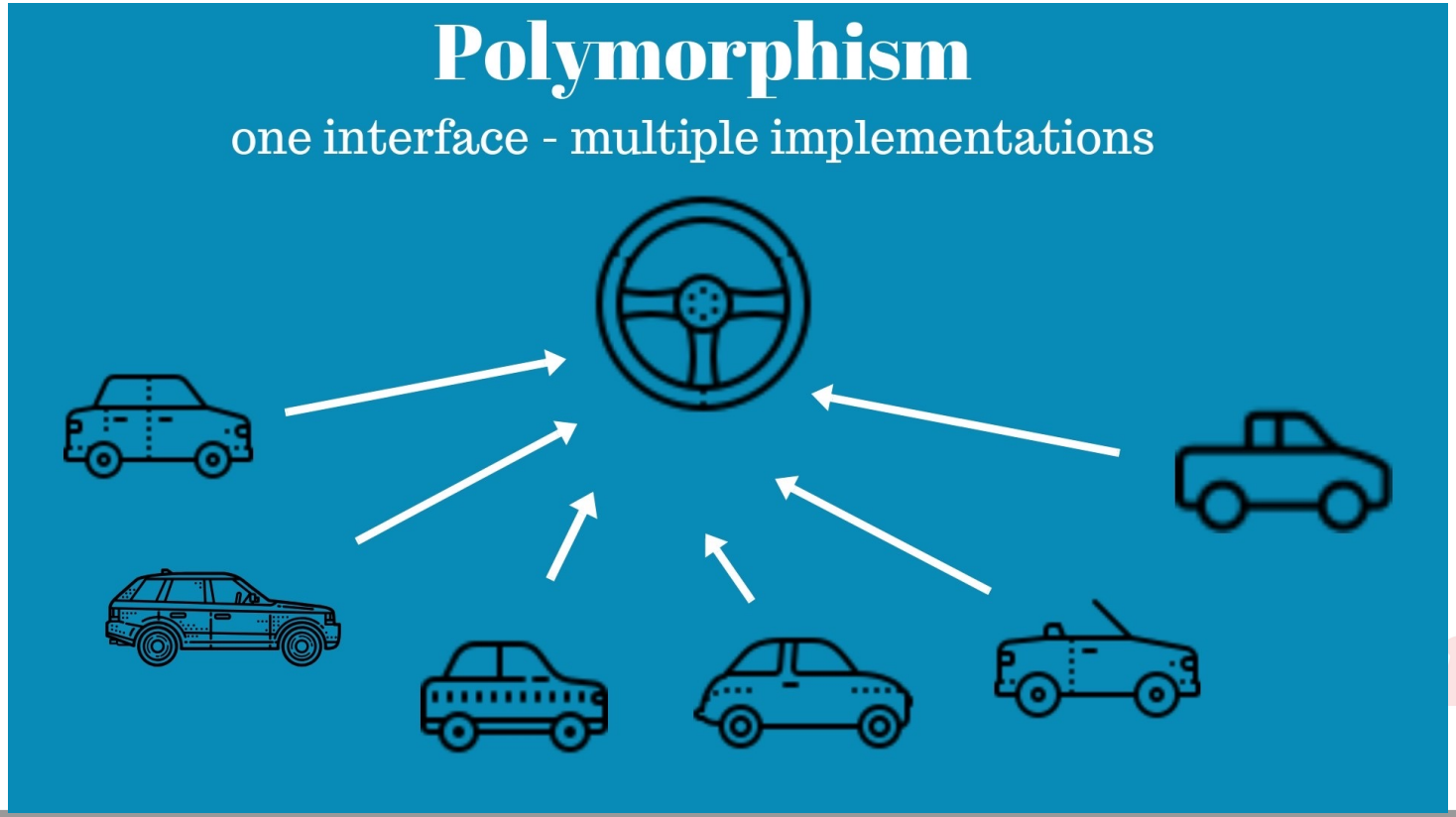
✓ Încapsulare

- ne concentrăm pe gruparea proprietăților și metodelor obiectului împreună într-o singură unitate.
- face codul mai modular și mai ușor de înțeles
- oferim securitate proprietăților și metodelor, hotărând cine le poate accesa.
- înseamnă ascunderea informațiilor.

✓ Abstractizare

- ne concentrăm pe ascunderea metodelor complexe și să arătăm utilizatorului doar lucrurile esențiale.
- face aplicațiile ușor de utilizat, ascunzând funcționarea complexă
- oferim securitate aplicației prin ascunderea părții de logică / implementării de utilizator.
- înseamnă ascunderea implementării

JavaScript – OOP– Polimorfism



JavaScript – OOP– Polimorfism

Moştenirea prezintă două aspecte esenţiale: **reutilizare de cod și polimorfism**.

Polimorfismul: este capacitatea unor entități de a lua forme diferite.

Etimologia polimorm: Poly = multe + Morfm = forma

Conform principiului Polimorfismului, metodele din clase diferite care fac lucruri similare ar trebui să aibă același nume.

*Pentru a implementa principiul polimorfismului, putem alege între clase abstracte și interfețe. (***Overloading și Overriding** *)*

JavaScript – OOP– Polimorfism

- ✓ Polimorfismul în Programarea Orientată pe Obiecte înseamnă că metoda dintr-o clasă se va comporta diferit de metoda cu acelaşi nume din altă clasă, ambele clase având un părinte comun.
- ✓ De exemplu, dacă spunem că câine şi pasăre sunt subclasele clasei animal, atunci amândouă au aceeaşi metodă: se deplasează. Aceste două metode, deşi au acelaşi nume, ar avea un rezultat complet diferit, deoarece pasărea zboară, iar câinele merge.
- ✓ Ca să implementăm polimorfismul, trebuie să existe cel puţin două clase copil.

JavaScript – OOP– Polimorfism

Polimorfismul poate fi de trei tipuri:

- ✓ **Adhoc Polymorphism -** înseamnă a schimba ceva de la o formă la alta pe loc.
- ✓ **Polimorfismul parametric** – mecanismul prin care putem defini o metodă cu același nume în aceași clasă (funcțiile trebuie să difere prin numărul și/sau tipul parametrilor – **Overloading: proprietați, metode**). Selecția funcției se realizează la compilate – legarea timpurie (**early binging**).
- ✓ **Polimorfismul de moștenire (**Subtype Polymorphism) – mecanismul prin care o metodă din clasa de bază este redefinită cu aceiași parametri în clasele derivate. Selecția funcției se va realiza la rulare – legarea întârziată (**late binding**, dynamic binding, runtime binding). Este abilitatea de a procesa obiectele în mod diferit, în funcție de tipul sau de clasa lor. [redefini metode pentru clasele derivate - **Overriding**]

JavaScript – OOP– Polimorfism

- ✓ **Adhoc Polymorphism**
  - ✓ **Operator Overloading**

    1 + 1 // 2

    1 + ‘1’ //11
  - ✓ **Coercion Type Polymorphism**

    22 == '22' // true
  - ✓ **Function Overloading**

    Ap6.html

JavaScript – OOP– Polimorfism

- ✓ Subtype Polymorphism - **Polimorfismul de moștenire**

  Ap7.html

  Ap8.html

Resurse

https://babeljs.io/

https://github.com/tc39/proposal-class-fields

https://codepen.io/bsehovac/pen/EMyWVv