

MINISTERUL EDUCATIEI AL REPUBLICII MOLDOVA

UNIVERSITATEA TEHNICA A MOLDOVEI

FACULTATEA CALCULATOARE INFORMATICA SI MICROELECTRONICA

DEPARTAMENTUL INGINERIA SOFTWARE SI AUTOMATICA

RAPORT

Lucrarea de laborator nr.1

Disciplina: Analiza, proiectarea si programarea orientata pe obiecte

Tema: Principiile OOP

A efectuat
st.gr. TI-153

Popusoi Victor

A controlat
asis., univ.

Pecari Mihail

Chisinau 2018

1 Scopul lucrarii de laborator

Implementarea celor patru principii de baza ale OOP, *incapsularea*, *abstractizarea*, *mostenirea*, *polimorfismul* in limbajul TypeScript.

2 Obiectivele lucrarii de laborator

Principiile programarii orientate pe obiecte:

- Abstractizarea;
- Incapsularea;
- Mostenirea;
- Polimosfismul.

3 Efectuarea lucrarii de laborator

3.1 Sarcinile propuse pentru efectuare lucrarii de laborator

Link-ul la repository *here*

1. Sa se implementeze principiile OOP in limbajul TypeScript

3.2 Realizarea lucrarii de laborator

Mai jos este aratat programul scris in limbajul TypeScript, in care au fost implementate cele patru principii de baza si anume: *incapsularea*, *abstractizarea*, *mostenirea* si *polimorfismul*

```
interface IVehicle {  
    nr(number: string)  
}  
  
abstract class Car implements IVehicle {  
    private model: string  
    private number: string  
    public engineModel(): string {  
        switch (this.model) {  
            case 'TDI':  
                return this.model = 'Engine ' + `${this.model}`  
            case 'CDI':  
                return this.model = 'Engine ' + `${this.model}`  
            default:  
                return this.model = 'Only TDI and CDI engine'  
        }  
    }  
}  
  
constructor(model: string) {  
    this.model = model
```

```

    }
    abstract nr(number: string)
}
class Engine extends Car {
    constructor(model: string) {
        super(model)
    }
    nr(number): void {
        console.log(`${this.engineModel()} have the number ${number}`)
    }
}
const engine: IVehicle = new Engine('TDI')
engine.nr('H22AM03737')

```

Pentru compilarea codului sau folosit comenzile: **tsc file.ts** si **node file.js** Comanda **tsc file.ts** este folosita pentru copilarea codului .ts si obtinerea fisierului .js dupa care este folosita comanda **node file.js** pentru a rula fisierul .js. Rezultatul programului de mai sus il puteti vedea in figura 3.1.

Abstracizarea este un concept incredibil de puternic în programarea orientată pe obiecte. Aceasta cuprinde ideea de a ascunde detaliile concrete ale implementării, oferind în același timp o definiție la nivel înalt a ceea ce ar trebui implementat. În exemplul de mai sus, putem vedea un caz de abstractizare. Interfața IVehicle creează stratul de abstractizare necesar pentru a gestiona conceptul de afisare a numarului motorului. Tipul Car este implementarea concreta ale acestei abstractizări. În următorul segment de cod, veți vedea un exemplu:

```

interface IVehicle {
    nr(number: string)
}
abstract class Car implements IVehicle {
    private model: string
    private number: string
    public engineModel(): string {
        switch (this.model) {
            case 'TDI':
                return this.model = 'Engine ' + `${this.model}`
            case 'CDI':
                return this.model = 'Engine ' + `${this.model}`
            default:
                return this.model = 'Only TDI and CDI engine '
        }
    }
    constructor(model: string) {

```

```

        this.model = model
    }
    abstract nr(number: string)
}

```

Conceptul de încapsulare în OOP ne permite să schimbăm funcționarea internă a unei clase fără a sparge nici unul dintre obiectele sale dependente. Acest lucru ar putea veni fie sub formă de membri privați, fie prin implementări de metode private. În următorul segment de cod, puteți vedea cum are loc ascunderea anumitor membri:

```

abstract class Car implements IVehicle {
    private model: string
    private number: string
    public engineModel(): string {
        switch (this.model) {
            case 'TDI':
                return this.model = 'Engine ' + `${this.model}`
            case 'CDI':
                return this.model = 'Engine ' + `${this.model}`
            default:
                return this.model = 'Only TDI and CDI engine '
        }
    }
    constructor(model: string) {
        this.model = model
    }
    abstract nr(number: string)
}

```

Moștenirea este capacitatea unei clase de a extinde funcționalitatea unei clase existente. Moștenirea creează o modalitate prin care obiectele multiple să împărtășească un set comun de cod și apoi să extindă sau să modifice acest lucru după cum este necesar pentru un anumit scop. De exemplu, clasa noastră Engine, care este foarte simplă cu câteva proprietăți în ea. În următorul cod, puteți vedea clasa Engine:

```

class Engine extends Car {
    constructor(model: string) {
        super(model)
    }
    nr(number): void {
        console.log(`${this.engineModel()} have the number ${number}`)
    }
}

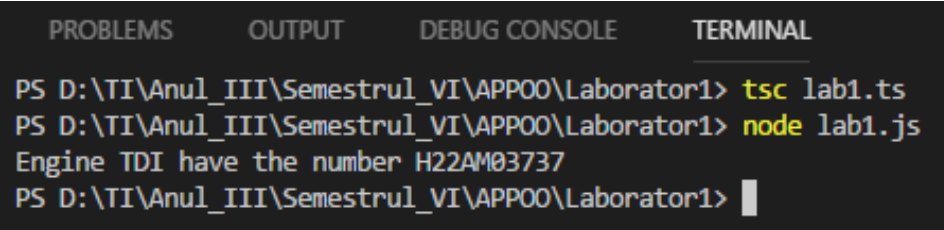
```

```
}  
}
```

Polimorfismul în definirea sa pură înseamnă să aibă multe forme. Când acest lucru este aplicat dezvoltării de software, acesta se aplică unei game largi de tehnici care ne permit să folosim o varietate de obiecte sau metode diferite pentru a efectua o sarcină. În limbajele POO, polimorfismul se referă la utilizarea supraîncărcării metodelor, supraîncărcarea operatorului și suprascrierea metodei.

```
const engine: IVehicle = new Engine('TDI')  
engine.nr('H22AM03737')
```

3.3 Imagini



```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL  
PS D:\TI\Anul_III\Semestrul_VI\APP00\Laborator1> tsc lab1.ts  
PS D:\TI\Anul_III\Semestrul_VI\APP00\Laborator1> node lab1.js  
Engine TDI have the number H22AM03737  
PS D:\TI\Anul_III\Semestrul_VI\APP00\Laborator1> |
```

Figure 3.1 – Final result

Concluzie

În aceasta lucrarea de laborator, au fost implementate cele 4 principii de baza ala Programarii Orientate pe Obiecte si anume, incapsularea, abstractizarea, mostenirea si polimorfismul. Unde incapsularea se refera la ascunderea informatiei, in procesul de abstractizare atentia este deci indreptata exclusiv spre aspectul exterior al obiectului, adica spre comportarea lui, ignorand implementarea acestei comportari. Cu alte cuvinte abstractizarea ne ajuta sa delimitam "CE face obiectul" de "CUM face obiectul ceea ce face". Prin mostenire se intelege reutilizarea codului, adica nu mai este nevoie pentru declararea unor anumite obiecte pentru fiecare clasa si prin polimorfism intelegem capacitatea unei entitati de a reactiona diferit in functie de starea sa.

Ideea POO este de a crea programele ca o colectie de obiecte, unități individuale de cod care interacționează unele cu altele, în loc de simple liste de instrucțiuni sau de apeluri de proceduri.

Bibliografie

1. TypeScript [Resursă electronică]. Regim de acces: <https://www.packtpub.com/mapt/>