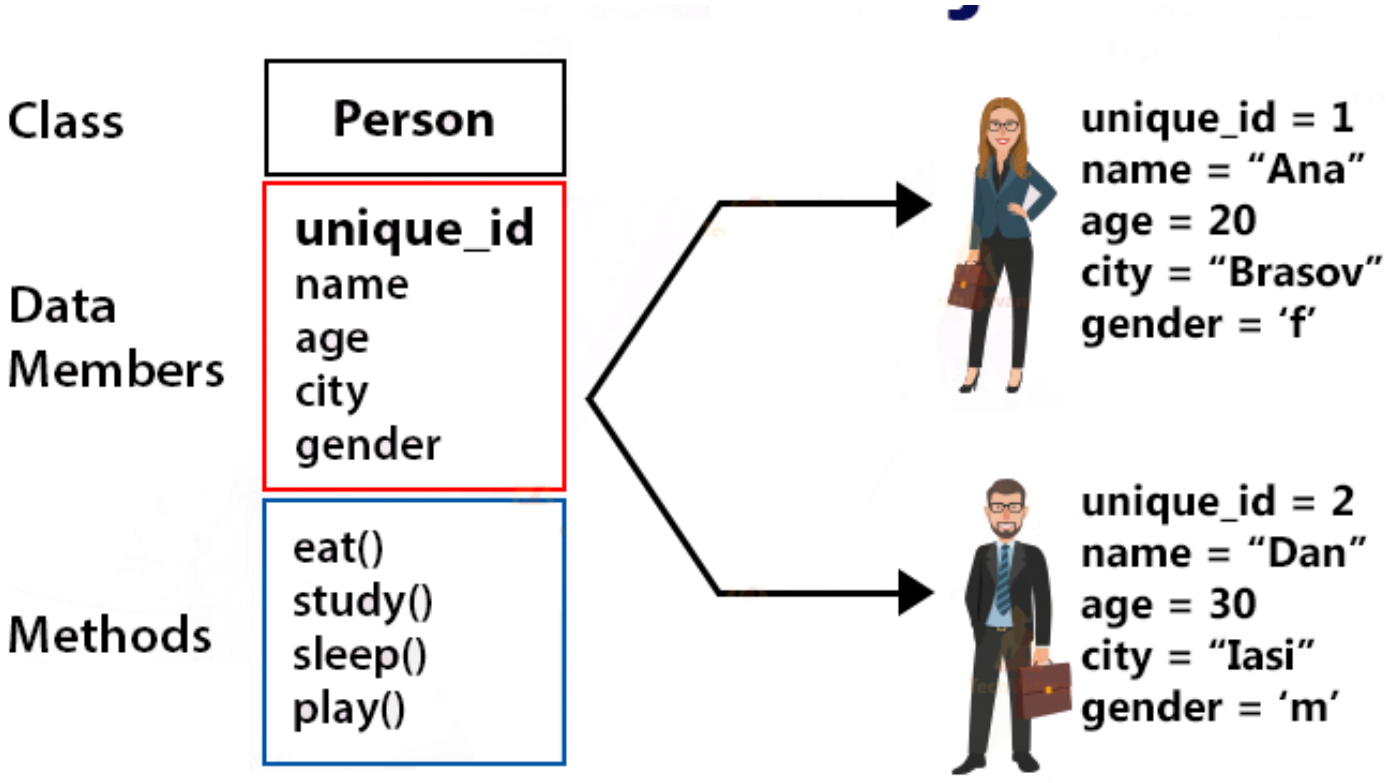
Curs 2: Clase si Obiecte

Cuprins

1. Reminder - Clase si Obiecte



Clasele sunt template-uri pentru obiecte, fiecare obiect este o **instanță** a o anumitei clase.

Cand se creeaza un obiect spunem ca se **instanțiază** un obiect al acelei clase.

Componente (se mai numesc **membri**):

- **câmpuri variabilă** care definesc **starea** obiectului
- **metode** care definesc **comportamentul**

2. Principiile de bază ale POO (Four Pillars of OOP)

1. Abstractizarea

(abstraction)

= principiul prin care:

- detaliile de implementare sunt ascunse
- este **expus doar un mecanism de nivel inalt**
- mecanismul intern nu este nevoie sa-l intelegem in totalitate cum functioneaza cata vreme este clar cum poate fi folosit
- de exemplu: o metoda care face intern lucruri complicate dar are parametri clar definiti si un efect clar definit = putem s-o folosim
- (efectul este ca nu este nevoie sa citim intreaga implementare a unor metode scrise de altcineva, ci putem pe baza parametrilor si documentatiei sa stim cum sa utilizam acele metode pentru a realiza ceea ce dorim)

(encapsulation)

= principiul prin care datele si operatiile sunt inglobate intr-un tot unitar:

(1) principiul ascunderii: datele obiectului sunt ascunse (private) pentru a nu putea fi accesate incorect în anumite prelucrări. Accesul si modificarea lor se face doar prin metode publice de tip **set/get**.

- (efectul este ca prin incapsulare **obiectul devine manager al propriei stari**, limiteaza si controleaza modul in care isi poate modifica starea)

(2) ascundem detaliile de implementare si **expunem doar strictul necesar pentru utilizare**

- (un exemplu este daca avem o metoda **publica** in care anumite sectiuni au un comportament foarte bine definit - e.g. deschis/ prelucrat fisiere - se pot muta acele sectiuni in alte metode **private** care sa fie chemate de acea metoda **publica** mai ales daca in restul metodei nu mai lucram cu fisiere)
- **!** ideea este diferita de cea de la abstractizare, aici dorim sa izolam cat mai bine lucrurile pentru a minimiza riscul efectelor secundare

3. Moștenirea

(inheritance)

= principiul prin care o clasă preia date și metode dintr-o clasă definită anterior, în scopul reutilizării codului creat anterior.

Daca avem nevoie de un obiect B care este doar putin diferit de un obiect A, putem sa reutilizam campurile/metodele lui A prin mostenirea lui A in B.

4. Polimorfismul

(polymorphism)

= principiul prin care un obiect poate avea comportament diferit prin utilizarea unei metode care are acelasi nume, dar implementare diferita fata de celelalte. Este de doua tipuri:

supraîncărcare (overloading / polimorfism static) = mecanismul prin care într-o clasă se pot defini două sau mai multe metode cu același nume, dar care au listele parametrilor diferite

suprascriere (overriding / polimorfism dinamic) = mecanismul prin care o subclasă rescrie o metodă a superclasei.

3. Declararea unei clase

```
    date membre/atribute

    metode membre //nu mai pot fi implementate în afara clasei!
}
```

Clasa este o implementare a unui tip de date de **referință** și poate fi privită ca un șablon pentru o categorie de obiecte.

4. Modificatorii de clasă *!diferiti! de modificatorii datelor membre:*

• **public**: clasa poate fi instanțiată și din afara pachetului său;

• **abstract**: clasa nu poate fi instanțiată (de obicei, deoarece conține cel puțin o metodă fără implementare – o metodă abstractă, dar în limbajul Java se poate declara ca fiind abstractă și o clasă care nu conține nicio metodă abstractă!);

• **final**: clasa nu mai poate fi extinsă.

Observație: Dacă nu exista modificatorul **public**, clasa are un acces implicit, adică poate fi instanțiată doar din interiorul pachetului în care a fost creată. (nu exista modificator **private** pentru clase)

5. Date membre

```
[modificatori] tip dataMembra = valoare initiala;
```

• Datele membre pot fi de orice tip, respectiv primitiv sau referință.

• Se declară ca orice variabilă locală, însă declararea poate fi însoțită și de **modificatori**.

• **Datele membre sunt inițializate cu valori nule** de tip (spre deosebire de variabilele locale, care nu sunt inițializate implicit, ci trebuie inițializate explicit)!

6. Modificatorii de acces pentru date membre *!diferiti! de modificatorii clasei*

• **public**: data membră poate fi accesată și din afara clasei, însă în conformitate cu principiul ascunderii (încapsulare) acestea sunt, de obicei, private;

• **protected**: data membră poate fi accesată din clasele aflate în același pachet sau din subclasele din ierarhia sa;

• **private**: data membră poate fi accesată doar din clasa din care face parte.

Observație: dacă nu este precizat niciun modificator de acces, atunci data membră respectivă are acces implicit, adică poate fi accesată doar din sursele aflate în același pachet.

7. Modificatorii static si final pentru date membre *!diferiti! de modificatorii clasei*

obiectelor.

• **final**: data membră poate fi doar inițializată, fără a mai putea fi modificată ulterior. Dacă data membră este un obiect, atunci nu i se poate modifica referința, dar conținutul său poate fi modificat!

Observație: Pentru o dată membră se pot combina mai mulți modificatori!

```
public static String facultate = "Informatica";
```

În concluzie, datele membre se împart în doua categorii:

• **date membre de instanță (date membre non-statice)** care se multiplică pentru fiecare obiect, alocându-se spațiu de memorie pentru fiecare în parte și fiind inițializate prin constructori;

• **date membre de clasă(date membre statice)** care sunt partajate de către toate obiectele, se alocă o singură dată și pot fi modificate de orice instanță (obiect) al clasei respective. Sunt utilizate pentru a defini date membre care nu depind de un anumit obiect (de exemplu, taxa TVA, curs valutar etc.) sau pentru a contoriza numărul de obiecte instanțiate. Datele membre statice nu se inițializează prin intermediul constructorilor!!!

```
class Persoana{
    private int IDPersoana;
    private String nume;
    private int varsta;
    private static String nationalitate = "română";
    private static int nrPersoane = 0;
    .........
}
```

8. Metode membre

```
    //corpul metodei
}
```

▪ Modificatorii unei metode membre sunt similari cu cei specifici datelor membre, la care se adaugă și modificatorul **abstract** prin care se declară o metodă fără implementare, care urmează să fie implementată în subclasele clasei respective.

▪ Utilizarea modificatorului **final** pentru o metodă membră împiedică redefinirea sa în subclasele clasei respective. De exemplu, o metodă care calculează TVA conține o formulă de calcul unică, care nu trebuie modificată/particularizată de către subclasele sale.

Observație: Parametrii unei metode sunt transmiși întotdeauna doar prin valoare!

Exemplu:

```
public class Test {
    static void modificare(int v[]) {
        v[0] = 100;
        v = new int[10];
        v[1] = 1000;
    }
    public static void main(String[] args) {
        int v[] = {1, 2, 3, 4, 5};
        modificare(v);
        System.out.println(Arrays.toString(v));
    }
}
```

După executare, se va afișa următorul tablou: **[100, 2, 3, 4, 5]**.

9. Metodele statice nu pot accesa date membre sau metode non-statice.

```
    String nume;
    static int nrPersoane;
    public String getNume() {
        return nume;
    }
    public void setNume(String nume) {
        this.nume = nume;
    }
    public static void afisareNumarPersoane(){
        System.out.println("Numar persoane: " + nrPersoane);
         //nu avem acces la nume
    }
}
```

10. Referința this

• Referința **this** reprezintă referința obiectului curent, respectiv a obiectului pentru care se accesează o dată membru sau o metodă membră.

• Referința **this** se poate utiliza în următoarele cazuri:

• pentru a accesa o dată membră sau pentru a apela o metodă:

```
this.nume="Popa Ion";
this.afișarePersoană();
```

• pentru a diferenția într-o metodă o dată membru de un parametru cu aceeași denumire:

```
public void setNume(String nume) {
    this.nume = nume;
}
```

11. Constructori

▪ Un constructor are numele identic cu cel al clasei și nu returnează nici o valoare.

▪ Un constructor nu poate fi **static**, **final** sau **abstract** .

▪ O clasă poate să conțină mai mulți constructori, prin mecanismul de supraîncărcare.

▪ Dacă într-o clasă nu este definit niciun constructor, atunci compilatorul va genera unul implicit (default), care va inițializa toate datele membre cu valorile nule de tip, mai puțin pe cele inițializate explicit!

12. Constructori cu si fara parametri

• **cu parametri**: inițializează datele membre cu valorile parametrilor

```
public Persoana(String nume, int varsta) {
    this.nume = nume;
    this.varsta = varsta;
}
```

• **fără parametri**: inițializează datele membre cu valori constante

```
public Persoana() {
    this.nume = "Popa Ion";
    this.varsta = 20;
}
```

▪ Pentru a apela constructorul cu argumente se poate utiliza referința **this**:

```
public Persoana() { this("Popa Ion",20); }
```

13. Constructor private

• De obicei, un constructor este **public**, însă există și situații în care acesta poate fi **private**:

• este necesar ca o clasă să nu fie instanțiată, de exemplu, dacă aceasta este o clasă de tip utilitar care conține doar date membre/metode statice (de exemplu, clasele java.lang.**Math** și java.util.**Arrays**);

• este necesar ca o clasă sa aibă o singură instanță (clasă **singleton <- design pattern**).

Exemplu: Considerăm o aplicație Java care modelează activitatea dintr-o organizație utilizând câte o clasă pentru fiecare încadrare specifică unui angajat, respectiv economist, director de departament, președinte etc. Evident, orice organizație are un singur președinte, deci clasa President care modelează acest rol trebuie să permită o singură instanțiere a sa!

Pentru a realiza o instanțiere unică a clasei President este necesară următoarea structură a clasei:

1. vom utiliza un câmp static care pentru a reține referința singurei instanțe a clasei;

```
class President {
        private static String name; //câmp de instanță
        private static President president; //1
        private President() { //2
                name = "Mr. John Smith";
        }
        public static President getPresident() { //3
                if (president == null)
                        president = new President();
                return president;
        }
        public static void showPresident(){
                System.out.println("President: " + name);
        }
}
public class Test{
        public static void main(String[] args) {
                President p = President.getPresident();//p=123
                President q = President.getPresident();//q=123
                System.out.println(p == q); //true
        }
}
```

▪ Se observă faptul că cele două referințe **p** și **q** sunt egale, iar singura instanță a clasei este creată doar în momentul în care aceasta este solicitată, adică în momentul în care este apelată metoda factory **getPresident()**. În acest caz spunem că se realizează o **instanțiere târzie (lazy initialization)**.

Observație: **În limbajul Java nu există constructor de copiere!** Evident, o clasă poată să conțină un constructor având ca parametru un obiect al clasei respective, în scopul de a copia în obiectul curent datele membre ale obiectului transmis ca parametru. Totuși, acest constructor nu va fi apelat automat în cazurile în care se apelează un constructor de copiere în alte limbaje orientate obiect (de exemplu, în limbajul C++).

▪ Declararea obiectului presupune definirea unei variabile alocată în zona de memorie stivă care va reține adresa obiectului după ce acesta este instanțiat. Dacă declararea obiectului se realizează local, în cadrul unei metode, atunci inițializarea sa cu null este obligatorie.

```
Persoana p = null;
```

▪ Instanțierea obiectului presupune alocarea unei zone de memorie HEAP necesară pentru a stoca membri obiectului și apelul unui constructor pentru a inițializa datele membre ale obiectului. Alocarea zonei de memorie HEAP se realizează folosind operatorul new care returnează adresa de memorie alocată sau null dacă alocarea nu s-a realizat cu succes.

```
p = new Persoana(nume, vârsta);
```

Observație: Un obiect poate fi instanțiat și în momentul declarării sale!

```
Persoana p = new Persoana(nume, vârsta);
```

• Funcționalitatea obiectului este asigurată de setul metodelor publice, astfel, după instanțiere sa, metodele membre publice pot fi apelate prin intermediul operatorului de accesare.

```
p.setNume("Popescu Ion");

System.out.println(p.getNume());
```

Observație: O data membră/metodă statică poate fi apelată și cu o referință null!

```
Persoana p = null;

p.afisareNumarPersoane();
```

• Distrugerea obiectului presupune eliberarea zonei de memorie alocată la instanțiere. Operația în sine, în limbajul Java, se realizează automat. Practic, mașina virtuală Java conține procesul Garbage Collection care conține un fir de executare dedicat, cu o prioritate scăzută, denumit **Garbage Collector (GC)**. Acesta scanează memoria și verifică dacă zonă de memorie mai este utilizată sau nu, marcând zonele nefolosite. Ulterior, zonele de memorie marcate sunt eliberate, respectiv sunt raportate ca fiind libere, și, eventual, se realizează o compactare a memoriei.

Un obiect devine eligibil pentru Garbage Collector în următoarele situații:

- nu mai există nicio referință, directă sau indirectă, spre obiectul respectiv;
- obiectul a fost creat în interiorul unui bloc (local) și executarea blocului respectiv s-a încheiat;
- dacă un obiect container conține o referință spre un alt obiect și obiectul container este devine null.

Înainte de a distruge un obiect, Garbage Collector apelează metoda **finalize** pentru a-i oferi obiectului r ctiv posibilitatea de a mai executa un set de acțiuni.

Collector folosind *System.gc()* sau *Runtime.getRuntime().gc()* !