# Programarea orientată pe obiecte

SERIA CB

## CLASE INTERNE ÎN METODE

O clasă se poate defini înăuntrul unei metode astfel:

```
class Extern {
    private String message = "Extern";

    void metoda() {
        class Intern {
            public void afiseazaExtern() {
                System.out.println("message = " + message);
            }
        }
        Intern in = new Intern();
        in.afiseazaExtern();
    }
}
```

---

```
class Extern {
    private String x = "Extern";

    void metoda() {
        final String z = "variabila locala";

        class Intern {
            public void afiseazaExtern() {
                System.out.println("x = " + x);
                System.out.println("z = " + z);
            }
        }
    }
}
```

Clasa `Intern` trebuie instanțiată în metoda în care este definită, deoarece vizibilitatea ei este la nivel de metodă, la fel ca a oricărei alte variabile declarate în metodă.

O clasă internă poate accesa orice membru al clasei externe.

O clasă internă **NU** poate accesa variabilele locale ale metodei în care este definită! **Variabilele locale sunt salvate pe stivă** și există atâta timp cât există și metoda.

Când execuția metodei se încheie, *stiva* este eliberată. Se poate întâmpla ca atunci când metoda ia sfârșit, obiectul din clasa internă să mai existe pe *heap*. Din cauza diferențelor de lungime de viață, instanțele claselor interne nu pot accesa variabilele locale ale metodelor în care au fost definite.

Acest lucru este posibil doar dacă se folosește modificatorul **`final`**.

La fel ca orice variabilă locală, clasele definite în metode **NU** pot avea modificatorii `public`, `private`, `protected`, `static` etc. Se pot folosi doar `abstract` și `final`, dar nu amândoi în același timp.

***O clasă definită într-o metodă statică are acces doar la membrii statici ai clasei externe.***

## CLASE INTERNE ANONIME

O clasă internă anonimă este o clasă fără nume. **Clasele interne anonime pot fi definite atât în metode, cât și în argumentul unei metode**.

```
class A {
    public void metoda() {
        System.out.println("Aceasta este o metoda");
    }
}
class B{
    A a = new A() {
        public void metoda() {
            System.out.println("metoda anonima");
        }
    };
}
```

O implementare anonimă a unei interfețe se poate face astfel:

```
interface A {
    public void metoda();
}
class B {
    A c = new A() {
        public void metoda() {
            System.out.println("implementare anonima");
        }
    };
}
```

- O clasă internă anonimă poate **implementa o singură interfață**.
- O clasă internă anonimă **nu poate extinde o clasă și implementa o interfață în același timp**!
- Dacă o clasă internă anonimă este o subclasă a unei clase, automat devine o implementare a unei interfețe implementate de superclasă.

Pentru interfața `Runnable`, următorul exemplu este greșit:

```
Runnable r = new Runnable();
```

Este corectă implementarea interfeței într-o instanță anonimă:

```
Runnable r = new Runnable() {
    public void metoda() { }
};
```

---

Motivul pentru care nu pot accesa o variabilă locală nefinală este că instanța locală a clasei rămâne în memorie după încheierea metodei. La încheierea metodei, variabilele locale devin *out-of-scope*, deci ar fi nevoie de o copie a lor.

*Dacă variabilele nu ar fi finale, copia lor din metodă s-ar putea modifica, pe când cea din instanța locală a clasei **nu**, ajungând astfel la o desincronizare a valorilor variabilelor.*

În clasa B este creat un obiect de tipul A care nu are nume de clasă, dar este o subclasă a lui A. Aceasta este o ***declarare anonimă.***

Metoda `metoda()` este suprascrisă în definiția anonimă. Tocmai acesta este scopul claselor anonime – *de a suprascrie metode ale claselor inițiale sau de a implementa metode ale unei interfețe.*

Definiția noii instanțe trebuie să se încheie obligatoriu cu semnul de punctuație ";".

## CLASE INTERNE ANONIME CA ARGUMENT

O clasă internă anonimă poate fi definită ca argument al unei metode:

```
class C {
    void exemplu() {
        B b = new B();
        b.executa(new A() {
            public void metoda() {
                System.out.println("Execut metoda");
            }
        });
    }
}
interface A {
    void metoda();
}
class B {
    void executa(A a) { }
}
```

## MOȘTENIREA CLASELOR INTERNE

La moștenirea claselor interne, constructorul clasei derivate trebuie să se atașeze de un obiect al clasei externe.

```
class Extern {
    class Intern {
        public void metoda() {
            System.out.println("Clasa interna");
        }
    }
}

class Subclasa extends Extern.Intern {
    Subclasa(Extern ext) {
        ext.super();
    }
}

public class Test {
    public static void main(String[] args) {
        Extern ext = new Extern();
        Subclasa s = new Subclasa(ext);
        s.metoda();
    }
}
```

# APLICAȚII:

## Test rapid de gripă și covid

În plin sezon de gripe (de tip A și tip B) și COVID, aveți nevoie să implementați o soluție rapidă pentru detectarea tipului de gripă în funcție de secreția nazală colectată.

În cadrul acestui link [ 1 ] vă sunt puse la dispoziție următoarele fișiere:

- `Mucus.java` – conține clasa `Mucus`, care este folosită pentru instanțierea obiectelor în funcție de tipul gripei și un „procent de siguranță” (nu modificați acest fișier);
- `InfluenzaDetector.java` – conține clasa `InfluenzaDetector` ce are o singură metodă `checkAllFlyTypes`, care primește ca parametru un obiect de tip `Mucus`. Metoda încearcă instanțierea unor obiecte de tipul `InfluenzaA`, `InfluenzaB` și `COVID`, iar apoi afișează rezultatul prin apelul unor metodei `isInfected` apelată pe obiectele respective. Trebuie să adăugați înăuntrul metodei `checkAllFlyTypes` cele 3 clase interne corespunzătoare, împreună cu metodele aferente lor.
  - ✓ Fiecare din cele 3 clase va avea ca variabilă membru pe `mucus` de tipul `Mucus`
  - ✓ Un constructor cu un parametru de tipul `Mucus`, ce setează variabila membru a clasei `(mucus)` cu parametrul metodei.
  - ✓ Implementarea metodei `public String getFluName()`, ce returnează numele tipului de gripă `("INFLUENZA_A" sau "NOT INFLUENZA_A" / "INFLUENZA_B" sau "NOT INFLUENZA_B"/ "COVID" sau "NOT COVID" )` în funcție de `mucus.type.name()` – care poate fi `"INFLUENZA_A", "INFLUENZA_B"` sau `"COVID"` (din `enum`)
  - ✓ Implementarea metodei `public boolean isInfected()`, ce returnează `true/false` în funcție de tipul de mucus și de procentajul său comparativ cu threshold-ul corespunzător (`mucus.type.name()` și `mucus.percentage`)
- `Main.java` – instanțiază un obiect `InfluenzaDetector` și rulează mai multe teste. Puteți modifica fișierul pentru a include teste suplimentare.

## Manager de studenți cu conexiune la baza de date

Aveți nevoie de integrarea unui manager de studenți cu o bază de date. Pentru aceasta trebuie să respectați un template impus într-o interfață.

În cadrul acestui link[ 2 ] vă sunt puse la dispoziție următoarele fișiere:

- `IDatabase.java` – interfața template ce definește metode de `connect` și `disconnect` (ce nu primesc parametru) și, respectiv `insert`, `update` și `delete` (ce primesc un parametru de tip `Object`)
- `Student.java` – o clasă simplă ce poate instanția studenți cu nume și având o reprezentare de tip `toString`
- `StudentManager.java` – o clasă ale cărei obiecte vor stoca un obiect de tip `IDatabase`. Clasa conține un `ArrayList` de obiecte de tip `Student` și metode pentru `insert`, `update` și `delete`. Fiecare metodă modifică lista și comunică cu baza de date. De asemenea, clasa conține și o reprezentare de tip `toString`. La finalul clasei există și o metodă `runSomeTests()`, pe care o puteți modifica pentru a adăuga teste;

- `Main.java` - conține un schelet de cod pentru apelul metodei `runSomeTests()` pentru un manager de studenți. Adăugați implementarea unei clase anonime în care să modificați metodele din interfața `IDatabase` și modificați de asemenea metodele `insertStudent`, `deleteStudent` și `updateStudent` din clasa `StudentManager` corespunzător astfel încât să faceți codul să compileze.

Într-o implementare anonimă a lui `IDatabase`:

- Instanțiați un obiect de tipul `StudentManager`
  `StudentManager studentManager = new StudentManager(this);`

- Definiți metodele `connect` și `disconnect`

```
public void connect() {
    System.out.println("Connected to the database");
}
public void disconnect() {
    System.out.println("Disconnected from the database");
}
```

- Definți metoda `public void insert(Object object)`, care introduce un `Student` în baza de date
- Definiți metoda `public void update(Object object, Object newObject)` pentru `Student`
- Definiți metoda `public void delete(Object object)`, ce șterge un `Student` din baza de date

  *Faceți downcasting de la tipul `Object` la tipul `Student`. Preluați codul ce realizează aceste operații din metodele din clasa `StudentManager`.*

- Definiți metoda `afisare`:

```
public void afisare()
{
    System.out.println(studentManager);
}
```

Modificați metodele următoare în clasa `StudentManager`, astfel:

```
public void insertStudent(Student student) {
      this.database.connect();
      this.database.insert(student);
      this.database.disconnect();
}

public void updateStudent(Student student, Student newStudent)
{
      this.database.connect();
      this.database.update(student, newStudent);
      this.database.disconnect();
}
```

```
public void deleteStudent(Student student) {
        this.database.connect();
        this.database.delete(student);
        this.database.disconnect();
}
```

# Referințe:


1. https://github.com/ACS-POO-2CB/lab-public/tree/main/poo-lab07/src/inner_class/inside_method

2. https://github.com/ACS-POO-2CB/lab-public/tree/main/poo-lab07/src/inner_class/anonymous

3. Oana Balan, Mihai Dascalu. **Programarea orientata pe obiecte in Java**. Editura Politehnica Press, 2020