# PROGRAMARE ORIENTATĂ PE OBIECTE

# LABORATOR 12

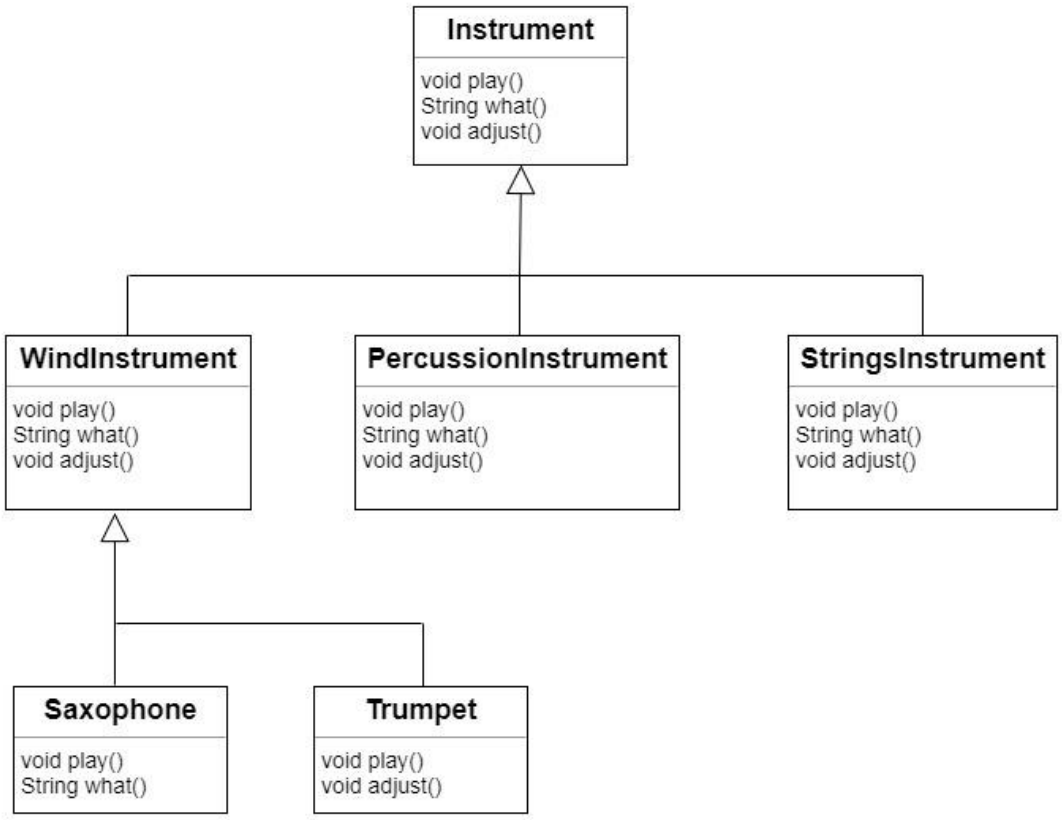
**Anul universitar 2023 – 2024**

**Semestrul II**

## *1 Noțiuni teoretice*

### *1.1 Clase abstracte*

Fie următorul exemplu (Thinking in Java) care propune o ierarhie de clase pentru a descrie o suită de instrumente muzicale, cu scopul demonstrării polimorfismului:




Clasa Instrument nu este instanţiată niciodată pentru că scopul său este de a stabili o interfaţă comună pentru toate clasele derivate. În același sens, metodele clasei de bază nu vor fi apelate niciodată. Apelarea lor este ceva greșit din punct de vedere conceptual.

### 1.1.1 Abstractizare

Abstractizarea este unul dintre cele 4 principii POO de bază (Abstractizare, Încapsulare, Moștenire, Polimorfism). Abstractizarea nu permite ca anumite caracteristici să fie vizibile în exterior. Acest lucru se referă la construirea unei interfețe comune, a unui șablon general pe care se bazează o anumită categorie de obiecte, fără a descrie explicit caracteristicile fiecaruia. Obiectele cunosc interfața comună pe care o au dar nu și cum este ea interpretată de fiecare obiect in parte. Acest lucru este realizat prin utilizarea claselor abstracte și a interfețelor.

### 1.1.2 Clase abstracte

Dorim să stabilim interfaţa comună pentru a putea crea funcţionalitate diferită pentru fiecare subtip și pentru a ști ce anume au clasele derivate în comun. O clasă cu caracteristicile enumerate mai sus se numește abstractă. Creăm o clasă abstractă atunci când dorim să:

- manipulăm un set de clase printr-o interfaţă comună
- reutilizăm o serie metode si membrii din această clasă in clasele derivate.

Metodele suprascrise în clasele derivate vor fi apelate folosind dynamic binding (late binding). Acesta este un mecanism prin care compilatorul, în momentul în care nu poate determina implementarea unei metode în avans, lasă la latitudinea JVM-ului (mașinii virtuale) alegerea implementării potrivite, în funcţie de tipul real al obiectului. Această legare a implementării de numele metodei la momentul execuţiei stă la baza polimorfismului. Nu există instanţe ale unei clase abstracte, aceasta exprimând doar un punct de pornire pentru definirea unor instrumente reale. De aceea, crearea unui obiect al unei clase abstracte este o eroare, compilatorul Java semnalând acest fapt.

### 1.1.3 Metode abstracte

Pentru a exprima faptul că o metodă este abstractă (adică se declară doar interfaţa ei, nu și implementarea), Java folosește cuvântul cheie abstract:

```
abstract void f();
```

O clasă care conţine metode abstracte este numită clasă abstractă. Dacă o clasă are una sau mai multe metode abstracte, atunci ea trebuie să conţină în definiţie cuvântul abstract.

Exemplu:

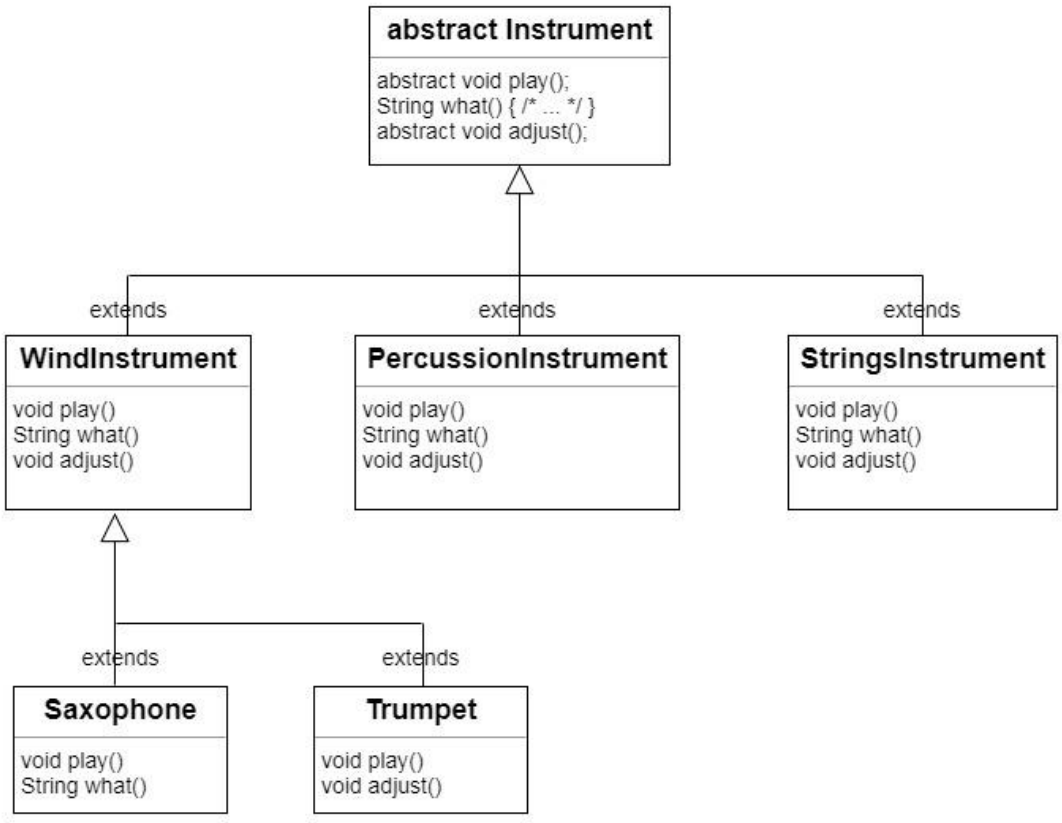
```
abstract class Instrument {
...
}
```

Deoarece o clasă abstractă este incompletă (există metode care nu sunt definite), crearea unui obiect de tipul clasei este împiedicată de compilator.

### 1.1.4 Clase abstracte în contextul moștenirii

O clasă care moștenește o clasă abstractă este ea însăși abstractă daca nu implementează toate metodele abstracte ale clasei de bază. Putem defini clase abstracte care moștenesc alte clase abstracte ș.a.m.d. O clasă care poate fi instanţiată (adică nu este abstractă) și care moștenește o clasă abstractă trebuie să implementeze (definească) toate metodele abstracte pe lanţul moștenirii (ale tuturor claselor abstracte care îi sunt "părinţi"). Este posibil să declarăm o clasă abstractă fără ca ea să aibă metode abstracte. Acest lucru este folositor când declarăm o clasă pentru care nu dorim instanţe (nu este corect conceptual să avem obiecte de tipul acelei clase, chiar dacă definiţia ei este completă).

Iată cum putem să modificăm exemplul instrument cu metode abstracte:




## *1.2 Interfețe*

### 1.2.1 Definire

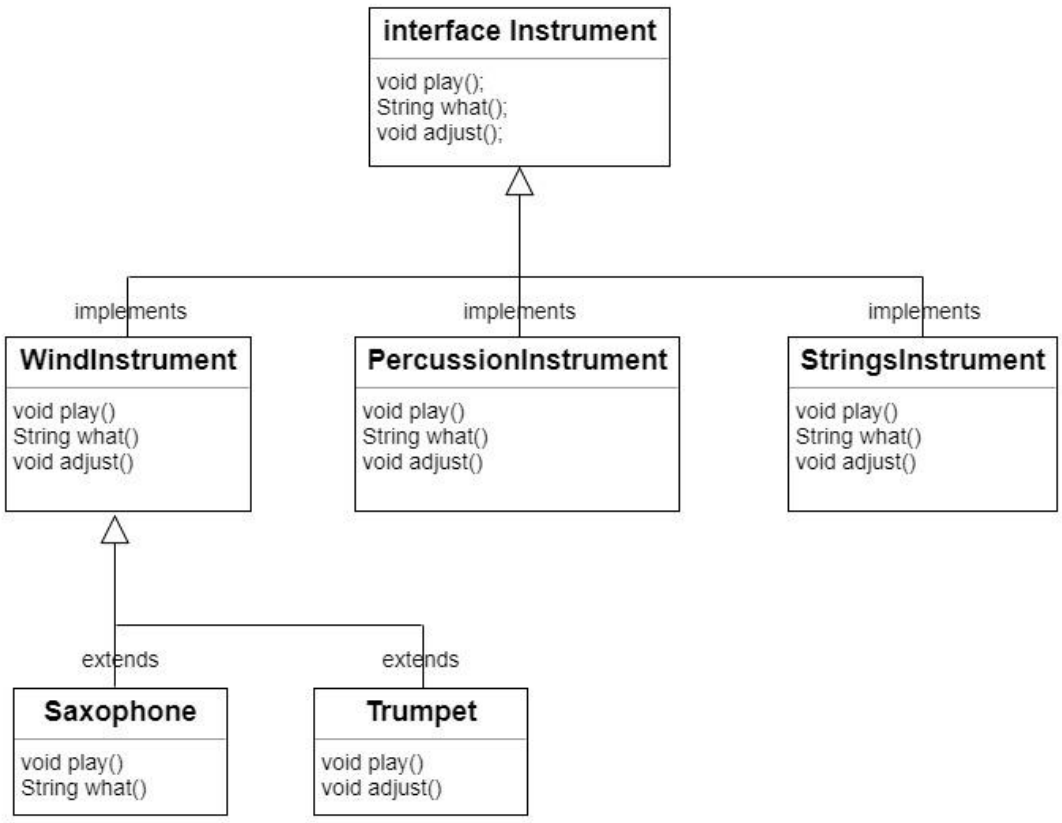
Interfeţele duc conceptul abstract un pas mai departe. Se poate considera că o interfață este o clasă abstractă pură: permite programatorului să stabilească o "formă" pentru o clasă (numele metodelor, lista

de argumente, valori întoarse), dar fară nicio implementare. O interfaţă poate conţine câmpuri dar acestea sunt în mod implicit static și final. Metodele declarate în interfață sunt în mod implicit public.

Interfaţa este folosită pentru a descrie un contract între clase: o clasă care implementează o interfaţă va implementa metodele definite în interfaţă. Astfel orice cod care folosește o anumită interfaţă știe ce metode pot fi apelate pentru aceasta.

Pentru a crea o interfaţă folosim cuvântul cheie interface în loc de class. La fel ca în cazul claselor, interfaţa poate fi declarată public doar dacă este definită într-un fișier cu același nume ca cel pe care îl dăm acesteia. Dacă o interfaţă nu este declarată public atunci specificatorul ei de acces este package-private. Pentru a defini o clasă care este conformă cu o interfaţă anume folosim cuvântul cheie implements. Această relaţie este asemănătoare cu moștenirea, cu diferența că nu se moștenește comportament, ci doar “interfaţa”. Pentru a defini o interfaţă care moștenește altă interfaţă folosim cuvântul cheie extends. Dupa ce o interfaţă a fost implementată, acea implementare devine o clasă obișnuită care poate fi extinsă prin moștenire.

Iata exemplul dat mai sus, modificat pentru a folosi interfeţe:



```
interface Instrument {

    // Compile-time constant:
    int FIELD = 5; // static & final

    // Cannot have method definitions:
    void play(); // Automatically public

    String what();

    void adjust();
}

class WindInstrument implements Instrument {

    public void play() {
        System.out.println("WindInstrument.play()");
    }

    public String what() {
        return "WindInstrument";
    }

    public void adjust() {
    }
}

class Trumpet extends WindInstrument {

    public void play() {
        System.out.println("Trumpet.play()");
    }

    public void adjust() {
        System.out.println("Trumpet.adjust()");
    }
}
```

### 1.2.2 Moștenire multiplă în Java

Interfaţa nu este doar o formă “pură” a unei clase abstracte, ci are un scop mult mai înalt. Deoarece o interfaţă nu specifică niciun fel de implementare (nu există nici un fel de spaţiu de stocare pentru o interfaţă) este normal să “combinăm” mai multe interfeţe. Acest lucru este folositor atunci când dorim să afirmăm că “X este un A, un B si un C”. Acest deziderat este moștenirea multiplă și, datorită faptului ca o singură entitate (A, B sau C) are implementare, nu apar problemele moștenirii multiple din C++.

## *1.3 Clase abstracte vs Interfețe*

Folosim o clasă abstractă atunci când vrem:

- să implementăm doar unele dintre metodele din clasă

- ca respectiva clasă să nu poată fi instanțiată

Folosim o interfață atunci când vrem:

- să avem doar o descriere a structurii, fără implementări
- ca interfața în cauză să fie folosită împreună cu alte interfețe în contextul moștenirii

# *2 Aplicații*

## *2.1 Firma IT*

Implementați o aplicație Java pentru gestionarea angajaților unei firme IT. Fiecare angajat are un nume, o vârstă, o vechime în firmă și un indicator pentru studii superioare. Există două tipuri de angajați: dezvoltatori și testeri. Fiecare angajat primește un salariu fix plus un bonus în funcție de vechime și de faptul dacă are sau nu studii superioare.

Clasa GestiuneAngajati este responsabilă pentru gestionarea listei de angajați și afișarea lor. Ea conține o metodă pentru adăugarea unui angajat în listă, o metodă pentru afișarea informațiilor despre angajați și o metodă pentru calculul salariului total al angajaților.

Programul trebuie să permită adăugarea unor angajați de ambele tipuri într-o listă și afișarea salariului total al acestora.

# 3 Rezolvări

## 3.1 Firma IT

```
package com.mycompany.firmait;

// Interfata Persoana
interface Persoana
{
    String getNume();
    int getVarsta();
}

interface Programator
{
    float bonus();
}

// Clasa abstracta Angajat
abstract class Angajat implements Persoana, Programator
{
    protected String nume;
    protected int varsta;
    protected int vechime;
    protected boolean studii;

    public Angajat(String nume, int varsta, int vechime, boolean studii)
    {
        this.nume = nume;
        this.varsta = varsta;
        this.vechime = vechime;
        this.studii = studii;
    }

    @Override
    public String getNume()
    {
        return nume;
    }

    @Override
    public int getVarsta()
    {
        return varsta;
    }

    @Override
    public float bonus()
    {
        if(this.studii == true)
        {
            return vechime * 10;
        }
        return 0;
```

```
    }

    public abstract double salariu();
}

// Clasa Dezvoltator
class Developer extends Angajat
{
    private final int h;

    public Developer(String nume, int varsta, int vechime, boolean prog, int h)
    {
        super(nume, varsta, vechime, prog);
        this.h = h;
    }

    @Override
    public double salariu() {
        return h * 20 + bonus();
    }
}

// Clasa Tester
class Tester extends Angajat {
    private final int bug;

    public Tester(String nume, int varsta, int vechime, boolean prog, int bug) {
        super(nume, varsta, vechime, prog);
        this.bug = bug;
    }

    @Override
    public double salariu() {
        return bug * 10 + bonus();
    }
}

// Clasa de gestionare a angajaților
class GestiuneAngajati {
    private final Angajat[] angajati;
    private int nr;

    public GestiuneAngajati(int MAX)
    {
        angajati = new Angajat[MAX];
        nr = 0;
    }

    public void adauga(Angajat angajat)
    {
        if (nr < angajati.length)
        {
            angajati[nr] = angajat;
```

```
            nr++;
        }
        else
        {
            System.out.println("Nu se mai pot adăuga angajați. Capacitatea maximă a fost
atinsă.");
        }
    }

    public void afis()
    {
        for (int i = 0; i < nr; i++) {
            System.out.println(angajati[i].getNume() + " - " + angajati[i].salariu());
        }
    }

    public double salariuTotal() {
        double salariuTotal = 0;
        for (int i = 0; i < nr; i++) {
            salariuTotal += angajati[i].salariu();
        }
        return salariuTotal;
    }
}

public class FirmaIT {

    public static void main(String[] args) {
        GestiuneAngajati gestiune = new GestiuneAngajati(10);

        Developer dev = new Developer("Alex", 30, 160, true, 10);
        Tester test = new Tester("Maria", 25, 50, true, 20);
        Angajat ang1 = new Tester("Ioana", 15, 15, false, 15);
        Angajat ang2 = new Developer("Sergiu", 10, 10, false, 20);
        Developer dev1 = new Developer("Alex", 30, 160, true, 10);
        Tester test1 = new Tester("Maria", 25, 50, true, 20);
        Angajat ang3 = new Tester("Ioana", 15, 15, false, 15);
        Angajat ang4 = new Developer("Sergiu", 10, 10, false, 20);
        Tester test2 = new Tester("Maria", 25, 50, true, 20);
        Angajat ang5 = new Tester("Ioana", 15, 15, false, 15);
        Angajat ang6 = new Developer("Sergiu", 10, 10, false, 20);

        gestiune.adauga(dev);
        gestiune.adauga(test);
        gestiune.adauga(ang1);
        gestiune.adauga(ang2);
        gestiune.adauga(dev);
        gestiune.adauga(test);
        gestiune.adauga(ang1);
        gestiune.adauga(ang2);
        gestiune.adauga(dev);
        gestiune.adauga(test);
        gestiune.adauga(ang1);
```

```
        gestiune.adauga(ang2);

        gestiune.afis();

        System.out.println("Salariu total: " + gestiune.salariuTotal());
    }
}
```

## *4 Teme*

Denumiți proiectele realizate de voi cu denumirea problemei și numele vostru. Exemplu: ***matrice_Bold Nicolae***.

### *4.1 Operații*

Creați 4 interfețe: Minus, Plus, Mult, Div care conțin câte o metodă aferentă numelui ce are ca argument un numar de tipul float.

Spre exemplu interfata Minus va declara metoda:

```
void minus(float value);
```

Scrieți clasa Operation care să implementeze toate aceste interfețe. Pe lânga metodele interfețelor implementate aceasta va conține:

- un câmp number de tipul float asupra căruia se vor efectua operațiile implementate
- metodata getter getNumber care va returna valoarea câmpului value
- un constructor care primește ca parametru valoarea inițială pentru campul value

Pentru cazul în care metoda div este apelată cu valoare 0 operația nu va fi efectuată și se va afișa mesajul “Division by 0 is not possible”.