

2010 - 2011

Programare Orientata spre Obiecte (*Object-Oriented Programming*)

a.k.a. Programare Obiect-Orientata

Titular curs: Eduard-Cristian Popovici

Suport curs: <http://electronica08.curs.ncit.pub.ro/course/view.php?id=113>

Suport curs vechi: <http://discipline.elcom.pub.ro/POO-Java/> si

<http://electronica07.curs.ncit.pub.ro/course/view.php?id=132>

Continut curs Programare Orientata spre Obiecte (in Java)

1. Introducere in abordarea orientata spre obiecte (OO)

- 1.1. Obiectul cursului si relatia cu alte cursuri
- 1.2. Evolutia catre abordarea OO
- 1.3. Caracteristicile si principiile abordarii OO
- 1.4. Scurta recapitulare a programarii procedurale/structurate (introducere in limbajul Java)

2. Orientarea spre obiecte in limbajul Java

- 2.1. Obiecte si clase. Metode (operatii) si campuri (attribute)
- 2.2. Particularitati Java. Clase de biblioteca Java (de uz general)
- 2.3. Clase si relatii intre clase. Asociere, delegare, agregare, compunere**
- 2.4. Generalizare, specializare si mostenire
- 2.5. Clase abstracte si interfete Java
- 2.6. Polimorfismul metodelor
- 2.7. Clase pentru interfete grafice (GUI) din biblioteca Java Swing

3. Programarea la nivel socket cu Java (pe platforma Java SE)

- 3.1. Clase pentru fluxuri de intrare-iesire (IO)
- 3.2. Introducere in Protocolul Internet (IP) si stiva de protocoale IP
- 3.3. Socketuri flux (TCP) Java.
- 3.4. Clase Java pentru programe multifilare. Servere TCP multifilare
- 3.5. Socketuri datagrama (UDP) Java

2. Orientarea spre obiecte in limbajul Java

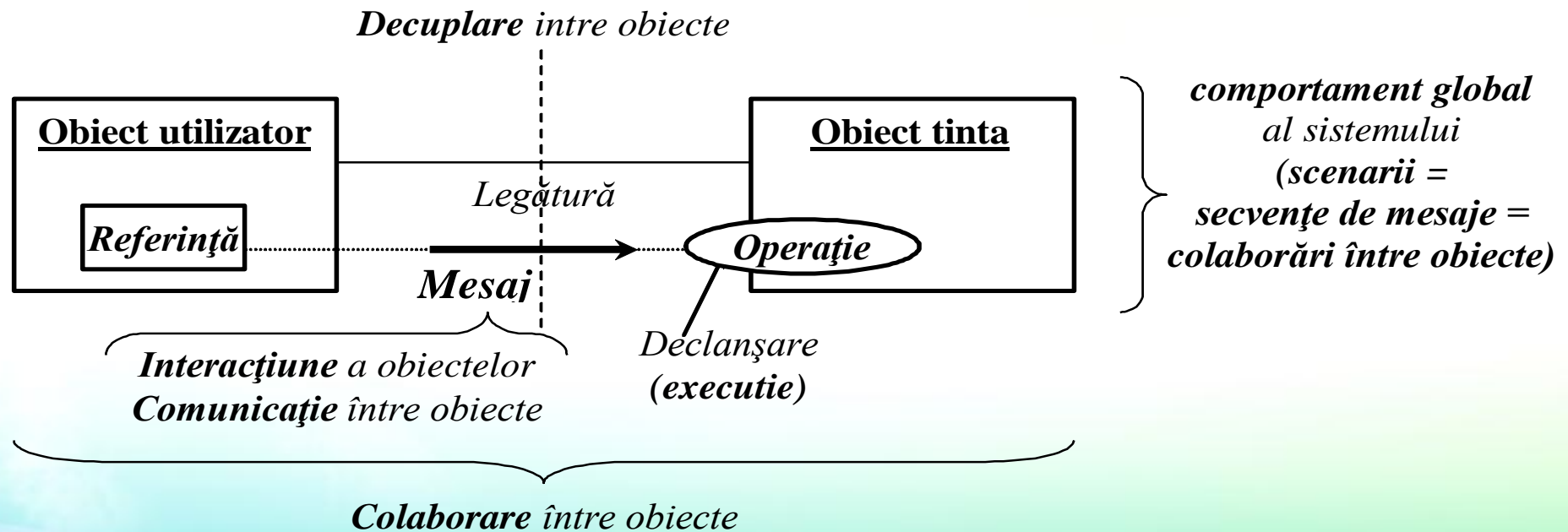
2.3. Clase si relatii intre clase.

Asociere, delegare, agregare, compunere

Relatiile cele mai simple dintre clase - asocierile

Asocierile dintre clase sunt **abstractizari ale legaturilor** dintre obiecte

- intre un obiect **utilizator** si un obiect **tinta**
 - se realizeaza o **legatura dinamica**
 - printr-o **referinta catre** obiectul **tinta detinuta de** obiectul **utilizator**
 - si **apelul** unei **metode** a obiectului **tinta** (*a.k.a.* **trimitere de mesaj**)



Exemplu de clasa tinta

```
public class Point {  
    // atribute (variabile membru)  
    private int x;  
    private int y;  
  
    // operatie care initializeaza attributele = constructor Java  
    public Point(int abscisa, int ordonata) {  
        x = abscisa;  
        y = ordonata;  
    }  
  
    // operatii care modifica attributele = metode (functii membru)  
    public void moveTo(int abscisaNoua, int ordonataNoua) {  
        x = abscisaNoua;  
        y = ordonataNoua;  
    }  
  
    public void moveWith(int deplasareAbsc, int deplasareOrd) {  
        x = x + deplasareAbsc;  
        y = y + deplasareOrd;  
    }  
  
    // operatii prin care se obtin valorile atributelor = metode  
    public int getX() { return x; }  
    public int getY() { return y; }  
}
```

} Declaratii
(specificare)
atribute

} Semnaturi
(declaratii,
specificari)
operatii
+
Implementari
(corpuri)
operatii

2.3. Clase si relatii intre clase

Exemplu de clasa utilizator pentru clasa tinta anterioara

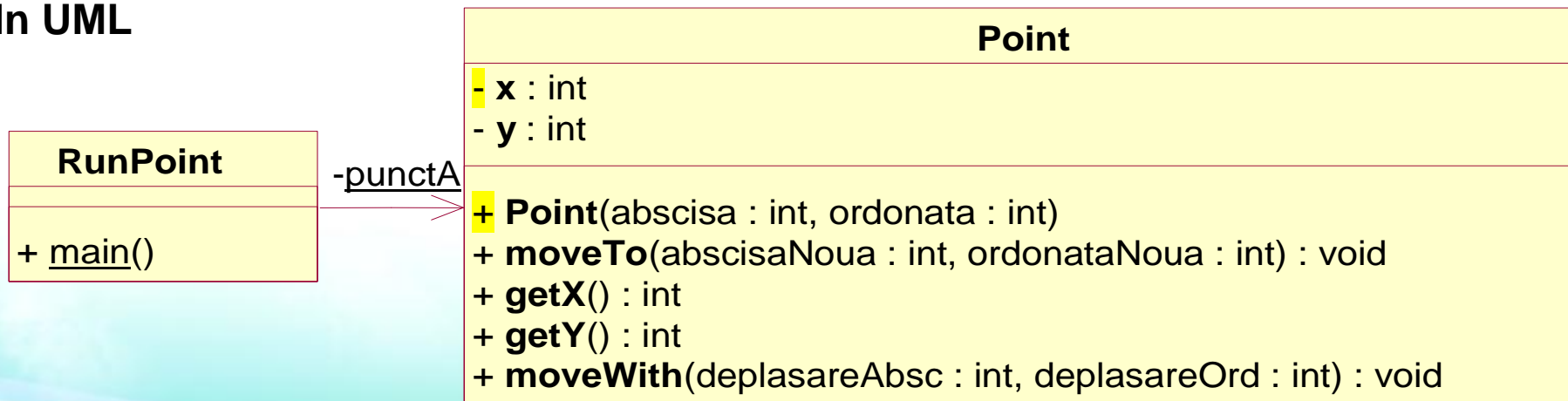
```
// clasa de test pentru clasa Point
public class RunPoint {
    private static Point punctA;           // atribut de tip Point

    public static void main(String[] args) { // declaratie metoda
        // corp metoda
        punctA = new Point(3, 4); // alocare si initializare atribut punctA

        punctA.moveTo(3, 5); // trimitere mesaj moveTo() catre punctA

        punctA.moveWith(3, 5); // trimitere mesaj moveWith() catre punctA
    }
}
```

In UML



Exemplu de clasa tinta

```
/**
 * Incapsuleaza informatiile si comportamentul unui Student.
 *
 */
public class Student {
    // Campuri (attribute) private (inaccesibile codurilor exterioare)
    private String nume;
    private String[] cursuri;
    private int[] rezultate;

    // Metode (operatii) publice (accesibile tuturor codurilor exterioare)
    // Metoda stabilire nume
    public void setNume(String n)
    {   nume = n;   }

    // Metoda stabilire cursuri
    public void setCursuri(String[] c)
    {   cursuri = c;   }

    // Metoda stabilire rezultate
    public void setRezultate(int[] r)
    {   rezultate = r;   }

    // Metoda obtinere nume
    public String getNume()
    {   return (nume);   }

    // Metoda obtinere cursuri
    public String[] getCursuri()
    {   return (cursuri);   }

    // Metoda obtinere rezultate
    public int[] getRezultate()
    {   return (rezultate);   }
}
```

Informatii ascunse = stare ascunsa (campuri private)

Interfata publica = servicii oferite (semnaturi metode)

Implementare ascunsa = comportament ascuns
(coduri interne ale metodelor)

2.3. Clase si relatii intre clase

Exemplu de clasa utilizator pentru clasa tinta anterioara

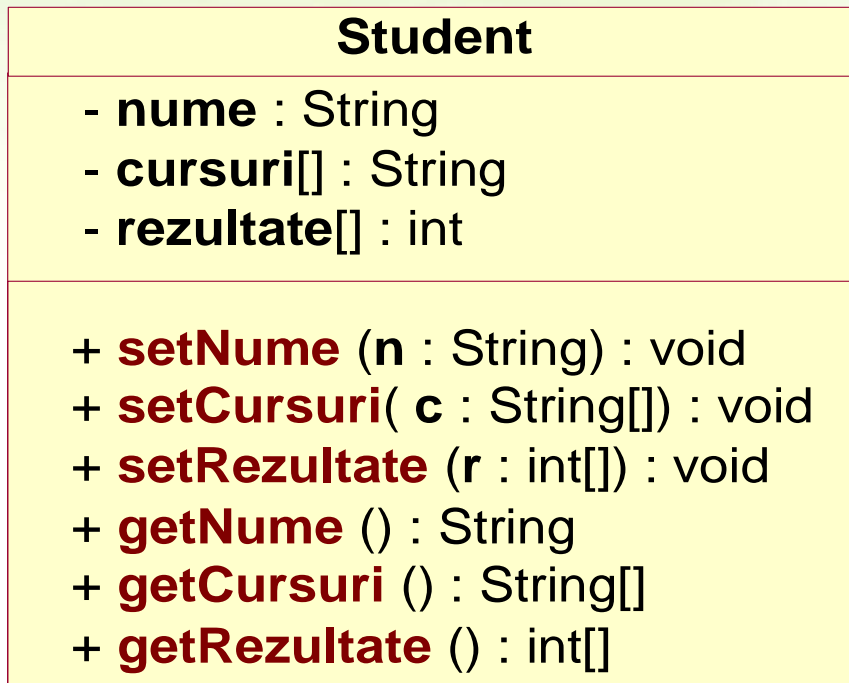
```
public class TestStudent {  
  
    // Metoda de test. Punct de intrare in program.  
    public static void main(String[] args) {  
  
        // Crearea unui nou Student, fara informatii  
        Student st1 = new Student();  
  
        // Initializarea campurilor noului obiect  
        st1.setNume("Xulescu Ygrec");  
        String[] crs = {"CID", "AMP"};  
        st1.setCursuri(crs);  
        int[] rez = {8, 9};  
        st1.setRezultate(rez);  
  
        // Utilizarea informatiilor privind Studentul  
        System.out.println("Studentul " + st1.getNume() + ":");  
        for (int i=0; i<rez.length; i++)  
            System.out.println("- are nota " + st1.getRezultate()[i]  
                + " la disciplina " + st1.getCursuri()[i]);  
    }  
}  
  
// Rezultatul: Studentul Xulescu Ygrec:  
// - are nota 8 la disciplina CID  
// - are nota 9 la disciplina AMP
```


2.3. Clase si relatii intre clase

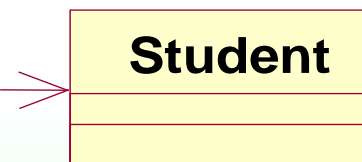
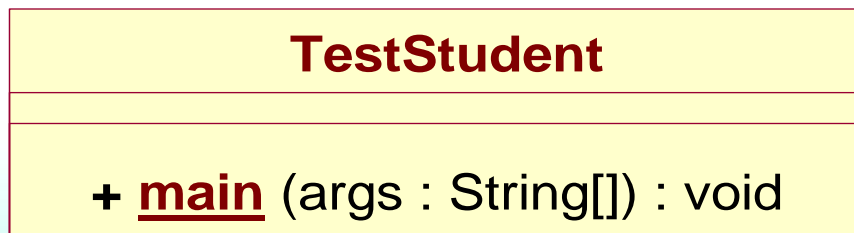
Exemplu de clasa utilizator si clasa tinta

In UML

Clasa tinta



Clasa utilizator



Clasa tinta



Exemplu de clasa tinta

Clasa tinta numita **Radio** care

- **simuleaza planurile** pentru **crearea unui obiect radio**

```
public class Radio {  
  
    // camp (atribut, variabila membru)  
    // definire tablou pentru asociere butoane cu frecvente  
    protected double[] stationNumber = new double[5];  
  
    // metoda (operatie, functie membru)  
    // definire asociere buton cu frecventa  
    public void setStationNumber(int index, double freq) {  
        stationNumber[index] = freq;  
    }  
  
    // metoda (operatie, functie membru)  
    // definire selectie frecventa  
    public void playStation(int index) {  
        System.out.println("Playing the station at " + stationNumber[index] + " Mhz");  
    }  
}
```

(explicatii la <http://discipline.elcom.pub.ro/POO-Java/Esenta POO - Obiecte Clase Incapsulare.pdf>
– adaptare dupa <http://www.developer.com/java/article.php/935351>)

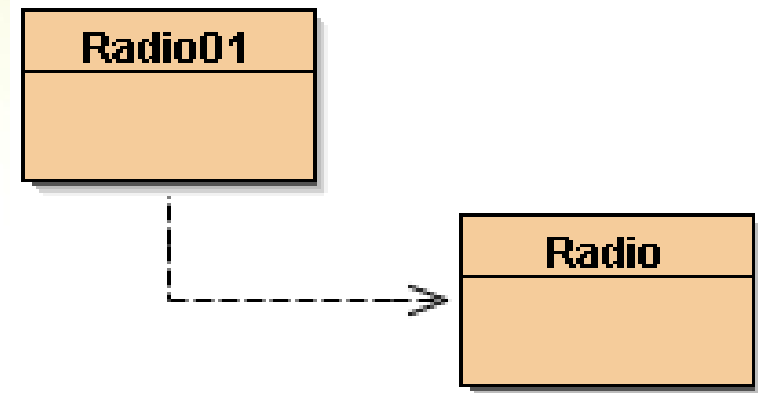
2.3. Clase si relatii intre clase

Exemplu de clasa utilizator pentru clasa tinta anterioara

Clasa utilizator numita **Radio01** care

- creeaza si foloseste un obiect al clasei Radio
- testeaza clasa Radio

```
public class Radio01 {  
  
    public static void main(String[] args){  
  
        // creare obiect  
        Radio myObjRef = new Radio();  
  
        // apel asociere buton cu frecventa  
        myObjRef.setStationNumber(3, 93.5);  
  
        // apel selectie frecventa  
        myObjRef.playStation(3);  
    }  
  
} // Rezultat: Playing the station at 93.5 MHz
```



2.3. Clase si relatii intre clase

Exemplu de clasa tinta – varianta 1 orientata procedural

Cum ar arata clasa **Radio** in varianta **orientata procedural**?

- in clasa **Radio001** tot codul este scris in metoda principala, main():

```
public class Radio001 {    // varianta strict procedurala (fara obiecte)

    public static void main(String[] args) { // tot codul in metoda main()

        // variabile locale
        // definire tablou pentru asociere butoane cu frecvente
        double[] stationNumber = new double[5];
        int index = 3;
        double freq = 93.5;

        // asociere buton cu frecventa
        stationNumber[index] = freq;

        // afisare selectie frecventa
        System.out.println("Playing the station at " + stationNumber[index] + " Mhz");
    }
} // Rezultat: Playing the station at 93.5 MHz
```

2.3. Clase si relatii intre clase

Exemplu de clasa tinta – varianta 2 orientata procedural

- in clasa **Radio002** metoda **main()** **delega 2 sarcini** catre **2 metode** (delegare / modularizare functionala):

```
public class Radio002 { // varianta strict functionala (fara obiecte)

    // delegare/modularizare functionala („orientare procedurala”)
    public static void main(String[] args) {
        // variabile locale
        double[] stationNumber = new double[5];
        int index = 3;
        double freq = 93.5;
        setStationNumber(stationNumber, index, freq); // delegare sarcina
        playStation(stationNumber, index);           // delegare sarcina
    }

    public static void setStationNumber(double[]stationNumber,
                                         int index, double freq){
        stationNumber[index] = freq;           // efectuare sarcina
    }

    public static void playStation(double[] stationNumber, int index){
        System.out.println("Playing the station at "
            + stationNumber[index] + " Mhz"); // efectuare sarcina
    }
} // Rezultat: Playing the station at 93.5 MHz
```

2.3. Clase si relatii intre clase



Exemplu de clasa tinta – varianta 3 orientata procedural

- in clasa **Radio003** apare un **atribut** declarat **static**, la nivelul clasei, **partajat de toate obiectele clasei** (modularizare la nivel de clasa):

```
public class Radio003 { // varianta cu camp (atribut) global (static)

    // camp static (la nivel de clasa, PARTAJAT de toate obiectele acesteia)
    private static double[] stationNumber = new double[5];

    private static void setStationNumber(int index, double freq){
        stationNumber[index] = freq; // utilizare camp al clasei (static)
    }

    private static void playStation(int index){
        System.out.println("Playing the station at " + stationNumber[index]
            + " Mhz"); // utilizare camp al clasei (static)
    }

    public static void main(String[] args) {
        int index = 3;
        double freq = 93.5;
        setStationNumber(index, freq) // delegare sarcina
        playStation(index); // delegare sarcina
    }

    // Rezultat: Playing the station at 93.5 MHz
}
```

2.3. Clase si relatii intre clase

Exemplu de clasa tinta

```
public class DatePersonale {  
  
    // Campuri ascunse  
    private String nume;  
    private String initiale;  
    private String prenume;  
    private int anNastere;  
  
    // Constructori  
    public DatePersonale(String n, String i, String p, int an) {  
        nume = new String(n); // copiere „hard”a obiectelor primitive parametri  
        initiale = new String(i); // adica se copiaza obiectul camp cu camp  
        prenume = new String(p); // nu doar referintele ca pana acum  
        anNastere = an;  
    }  
  
    // Interfata publica si implementarea ascunsa  
    public String getNum() { return (nume); }  
    public String getPrenume() { return (prenume); }  
    public int getAnNastere() { return (anNastere); }  
  
    public String toString() { // forma „String” a campurilor obiectului  
        return (nume + " " + initiale + " " + prenume + " (" + anNastere + ")");  
    }  
}
```

Exemplu de clasa tinta

```
public class SituatieCurs {  
  
    // Campuri ascunse  
    private int nota = 0; // initializare implicita  
    private String denumire;  
  
    // Constructor  
    public SituatieCurs(String d) { denumire = new String(d); } // copiere  
    // „hard”, se initializeaza doar denumire  
  
    // Interfata publica si implementarea ascunsa  
    public void notare(int n) { nota = n; } // se adauga nota  
  
    public int nota() { return(nota); } // se returneaza nota  
  
    public String toString() { // forma „String” a campurilor  
        if (nota==0)  
            return ("Disciplina " + denumire + " nu a fost notata");  
        else  
            return("Rezultat la disciplina " + denumire + ": " + nota);  
    }  
}
```


2.3. Clase si relatii intre clase

Exemplu de clasa utilizator pentru clasele tinta anterioare

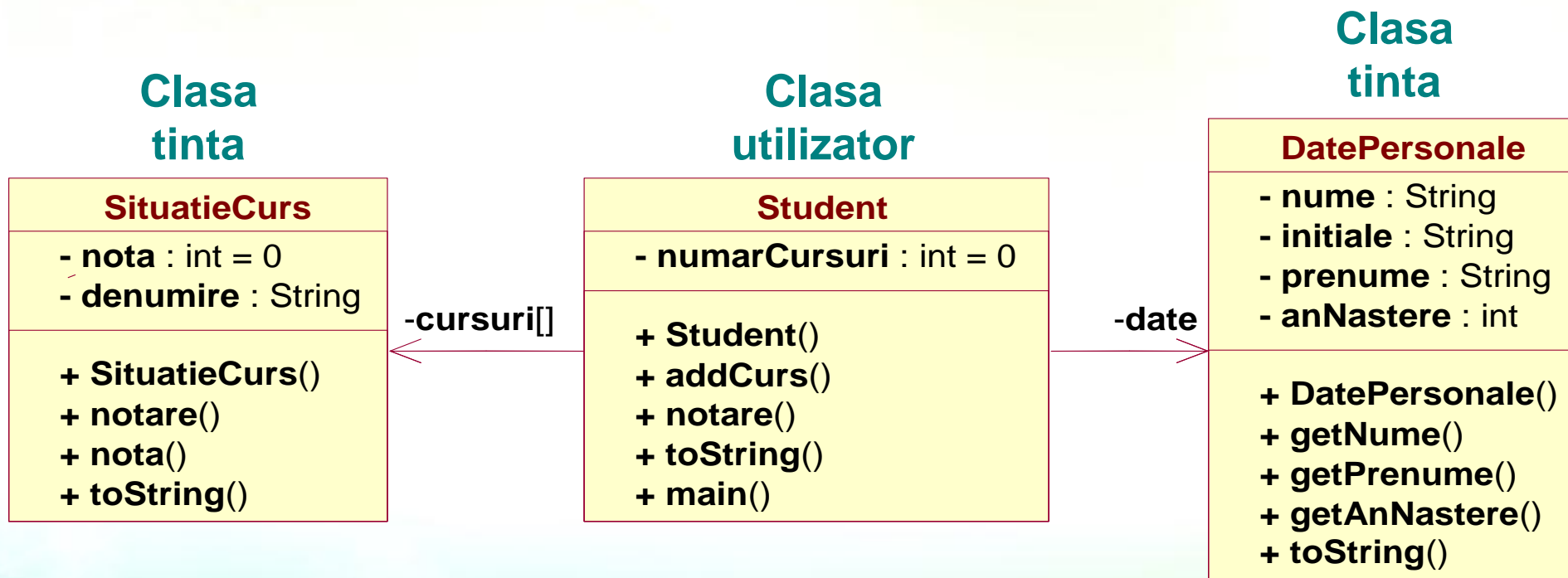
```
public class Student { // rescrisa pentru a putea utiliza clasele anterioare
    // Campuri ascuse
    private DatePersonale date;
    private SituatieCurs[] cursuri;
    private int numarCursuri = 0; // initializare implicita
    // Constructori
    public Student(String nume, String initiale, String prenume, int anNastere){
        date = new DatePersonale(nume, initiale, prenume, anNastere);
        cursuri = new SituatieCurs[10]; // se initializeaza doar date si cursuri
    }
    // Interfata publica si implementarea ascunsa
    public void addCurs(String nume) { // se adauga un nou curs
        cursuri[numarCursuri++] = new SituatieCurs(nume);
    }
    public void notare(int numarCurs, int nota) {
        cursuri[numarCurs].notare(nota); // se adauga nota cursului specificat
    }
    public String toString() { // forma „String” a campurilor
        String s = "Studentul " + date + " are urmatoarele rezultate:\n";
        for (int i=0; i<numarCursuri; i++)
            s = s + cursuri[i].toString() + "\n"; // delegare orientata spre obiecte
        return (s);
    }
}
```

2.3. Clase si relatii intre clase

Exemplu de clasa utilizator pentru clasa tinta anterioara

In UML

- se observa ca **atributele** cursuri si date sunt **reprezentate** ca **asocieri intre clase**



Exemplu de clasa tinta

Implementarea clasei **Complex**

- in forma **carteziana** (atributele ascunse sunt **coordonatele carteziene**)

```
public class Complex {  
  
    private double real;           // partea reala (abscisa)  
    private double imag;         // partea imaginara (ordonata)  
  
    public Complex(double real, double imag) {  
        this.real = real;  
        this.imag = imag;  
    }  
  
    public double getReal() {      return this.real;    }  
    public double getImag() {     return this.imag;    }  
  
    public double getModul() {  
        return Math.sqrt(this.real*this.real + this.imag*this.imag);  
    }  
    public double getFaza() {  
        return Math.atan2(this.real, this.imag);  
    }  
}
```

Exemplu de clasa tinta

Implementarea clasei **Complex**

- in forma **polara** (atributele ascunse sunt **coordonatele polare**)

```
public class Complex {  
  
    private double modul;           // modulul (raza)  
    private double faza;           // faza (unghiul)  
  
    public Complex(double real, double imag) {  
        this.modul = Math.sqrt(real*real + imag*imag);  
        this.faza = Math.atan2(real, imag);  
    }  
  
    public double getReal() {  
        return this.modul*Math.cos(this.faza);  
    }  
    public double getImag() {  
        return this.modul*Math.sin(this.faza);  
    }  
  
    public double getModul() {      return this.modul;      }  
    public double getFaza() {      return this.faza;      }  
}
```

2.3. Clase si relatii intre clase

Exemplu de clasa utilizator pentru clasa tinta anterioara

Urmatorul cod Java va conduce la **acelasi rezultat**

- **indiferent de forma de implementare a clasei Complex**

```
public class TestComplex {  
    public static void main(String[] args){  
        Complex c1 = new Complex(2, -2);  
        System.out.println("Coordonatele carteziene: {"  
            + c1.getReal() + ", " + c1.getImag() + "}");  
        System.out.println("Coordonatele polare: {"  
            + c1.getModul() + ", " + c1.getFaza() + "}");  
    }  
}
```

- **deoarece specificatia publica este comuna**

```
public class Complex {  
    // Atributele sunt private (ascunse, inaccesibile din exteriorul clasei)  
    // Constructorul - initializeaza obiectele de tip Complex  
    public Complex(double real, double imag) { }  
    public double getReal() { } // Returneaza partea reala  
    public double getImag() { } // Returneaza partea imaginara  
    public double getModul() { } // Returneaza modulul  
    public double getFaza() { } // Returneaza faza  
}
```

Relatii intre clase – cazuri speciale

Asocierea are doua cazuri speciale

- **agregarea** (relatia de **subordonare** a unei clase **de catre o clasa numita agregat**)
- **compunerea** (forma puternica de agregare, de tip **intreg-parte**, in care **partile**, numite **componente**, **apartin strict intregului**, numit **compozit**)

Association

Objects are aware of one another so they can work together

Aggregation

1. Protects the integrity of the configuration
2. Functions as a single unit
3. Control through one object – propagation downward

Composition

Each part may only be a member of one aggregate object

Relatii intre clase – cazuri speciale

In UML

- **agregarea** se reprezinta prin **romb alb**
- **compunerea** se reprezinta prin **romb negru**

