

Laborator 5 – Programare Orientata pe Obiect

1. Consideratii teoretice despre mostenire:

Moștenire ea este un mecanism de reutilizare a codului. Prin mostenire clasă copil **extinde** o clasă deja existentă, clasa părinte. În terminologia Java, o clasă care este moștenită se numește părinte sau superclasă, iar noua clasă se numește copil sau subclasă. Ideea din spatele moștenirii în Java este că puteți crea noi clase care sunt construite pe baza claselor existente. Când moștenești dintr-o clasă existentă, puteți reutiliza metodele și câmpurile clasei părinte. Mai mult, puteți adăuga noi metode și câmpuri în clasa curentă.

Moștenirea reprezintă relația IS-A, cunoscută și sub numele de relație părinte-copil.

Constructorii **nu** se moștenesc și pot fi apelați doar în contextul unui constructor copil. Apelurile de constructor sunt înălțuite, ceea ce înseamnă că înainte de a se inițializa obiectul copil, mai întâi se va inițializa obiectul părinte. În cazul în care părintele este copil la rândul lui, se va inițializa părintele lui (până se va ajunge la părintele suprem – root).

În Java, clasele și membrii acestora (metode, variabile, clase interne) pot avea diverși specificatori de acces.

- specificatorul de acces **protected** - specifică faptul că membrul sau metoda respectivă poate fi accesată doar din cadrul clasei înseși sau din clasele derivate din această clasă.
- specificatorul de acces **private** - specifică faptul că membrul sau metoda respectivă poate fi accesată doar din cadrul **clasei** înseși, nu și din clasele derivate din această clasă.

Sintaxa moștenirii Java.

class Subclass-name extends Superclass-name

```
{  
  
    //methods and fields  
  
}
```

Cuvântul cheie **extinde** indică faptul că creați o nouă clasă care derivă dintr-o clasă existentă.

Suprascrierea metodelor. În orice limbaj de programare orientat obiect, overriding este o caracteristică care permite unei subclase sau unei clase copil să ofere o implementare specifică a unei metode care este deja furnizată de una dintre superclasele sau clasele părinte. Când o metodă dintr-o subclasă are același nume, aceiași parametri sau semnătură și același tip de returnare (sau subtip) ca o metodă din super-clasă sa, atunci se spune că metoda din subclasă suprascrive metoda din super –clasă.

Exemplu de overriding.

```
class Person {  
    String name;  
    void show()  
    {  
        System.out.println("information about a person"+toString());  
    }  
}
```

```

    }
    public String toString(){
        return "name: " +name;
    }
}

// Inherited class
class Student extends Person {
    int regNumber;

    @Override
    void show()
    {
        System.out.println("information about a student"+toString());
    }
    @Override
    public String toString(){
        return super.toString()+" , registration number:"+regNumber;
    }
}

// Driver class
class Main {
    public static void main(String[] args)
    {
        // If a Person type reference refers
        // to a Person object, then Person's
        // show is called
        Person obj1 = new Person();
        obj1.show();

        // If a Person type reference refers to a Student object Student's show() is called.
        //This is called RUN TIME POLYMORPHISM.
        Person obj2 = new Student();
        obj2.show();
    }
}

```

Metodele declarate **finale** nu pot fi suprascris. Dacă nu dorim ca o metodă să fie suprascrisă, o declarăm final.

```

class Person {
    .....
    // Can't be overridden
    final void show() {}
}

```

```

class Student extends Person {
.....
    // This would produce error
    void show() {}
}

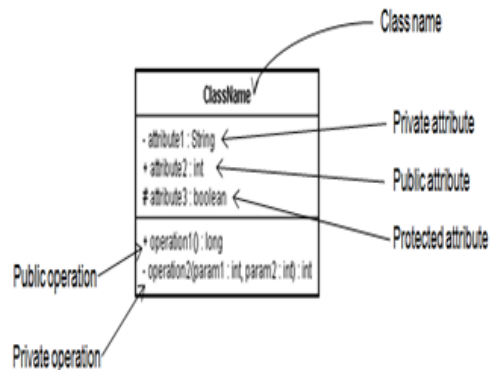
```

2. Diagrame UML – Diagrame de clasa

O clasă este reprezentată grafic în UML printr-un dreptunghi în care sunt specificate următoarele informații:

- Numele clasei:
 - Folosit pentru a distinge clasa de alte clase
 - Numele clasei apare în prima partiție.
 - Simplu sau calificat (prefixat cu numele pachetului căruia îi aparține clasa)
- Atributele clasei
 - Atributele sunt afișate în a doua partiție.
 - Tipul de atribut este afișat după două puncte.
 - Atributele mapează variabilele membre (membrii de date) în cod
- Operațiile clasei
 - Operațiile sunt prezentate în a treia partiție.
 - Tipul returnat al unei metode este afișat după două puncte la sfârșitul semnăturii metodei
 - Tipul returnat al parametrilor metodei este afișat după două puncte după numele parametrului.
 - Operațiunile se mapează pe metodele clasei în cod

Figura de mai jos ilustrează reprezentarea în UML a unei clase.


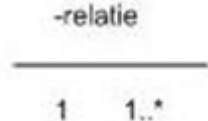






Membrii clasei (atribute și metode) au asociat un nivel de vizibilitate specific prin intermediul modificadorului de acces. Tabelul de mai jos prezintă modalitatea de reprezentare a nivelului de vizibilitate al unui membru al unei clase în diagrama UML.

Modificator	Symbol	Domeniu de vizibilitate
public	+	Vizibil oriunde în program; poate fi apelat de orice obiect din sistem
private	-	Vizibil la nivelul clasei în care a fost definit
protected	#	Vizibil la nivel de pachet și la nivel de subclase
package	~	Vizibil la nivel de pachet

O clasă poate fi implicată într-una sau mai multe **relații** cu alte clase. Intre doua clase pot exista urmatoarele tipuri de relatii:

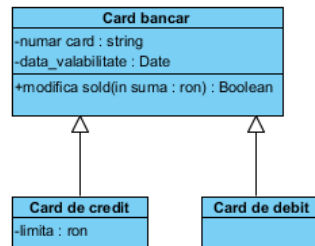
- Moștenire
- Asociere simplă
- Agregare
- Compoziție
- Dependență
- Realizare

Tip de relatie	Definitie	Reprezentare in UML
Mostenire	O relație de moștenire între două clase indică faptul că o clasă copil moștenește caracteristicile clasei părinte. Copilul are în plus atribute și operații pe lângă cele moștenite la părinții săi.	
Asociere	Modelează o conexiune semantică între clase (ex. un angajat lucrează pentru o organizație). O asociere poate avea un nume folosit pentru a descrie natura relației. Exemple de asociere: bidirecțională, unidirecțională, agregare, compoziției. Acest lucru poate fi specificat folosind multiplicitate (unu la unu, unul la mulți, mulți la mulți etc.).	
Agregarea	Agregarea este un tip special de asociere folosit pentru a modela o relație „întreg / parte”, în care o clasă reprezintă un lucru mai mare („întregul”), care constă din lucruri mai mici („părțile”). Agregarea indică faptul că o clasă părinte are elemente de tipul clasei copil. În agregare partea poate exista independent de întreg. De exemplu, dacă considerăm două clase: Portofel și Bani. Un portofel „conține” bani. Dar banii nu trebuie neapărat să aibă un portofel, deci între cele două clase avem o relație de agregare.	
Compozitia	Compoziția este o variație a agregării simple în care clasa copil există doar dacă există o instanță a clasei părinte. Un om are nevoie de o inimă pentru a trăi și o inimă are nevoie de un corp pentru a funcționa. Cu alte cuvinte, când clasele (entitățile) sunt dependente una de cealaltă și durata lor de viață este aceeași (dacă una moare, moare și cealaltă), atunci este o relație de compoziție	
Dependentă	Relația de dependență înseamnă că un obiect al unei clase ar putea folosi un obiect al altei clase în codul unei metode. Dacă obiectul nu este stocat în niciun câmp, atunci acesta este modelat ca o relație de dependență. De exemplu, clasa Person ar putea avea o metodă hasRead cu un parametru Book care returnează true dacă persoana a citit cartea (poate verificând o bază de date).	
Realizare	O relație de realizare este o relație semantică între clasificatori (de exemplu relația între o clasă care implementează o	

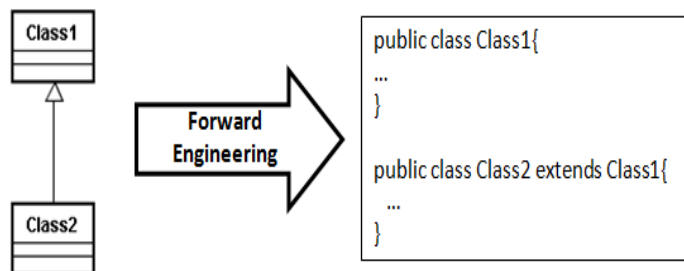
	interfata, clase, interfețe) în care un clasificator specifică un contract pe care un alt clasificator îl garantează pentru realizare	
--	---	--

a) **Exemplu de mostenire:**

Clasa **Card bancar** este mostenita de clasele **Card de Debit** si **Card de Credit**.



Cum se transcrie o relatie de mostenire la nivel de cod:



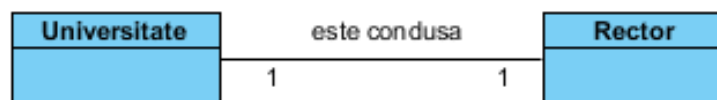
b) **Exemplu de relatii de asociere intre doua clase.**

Intr-o relatie de asociere, multiplicitatea definește câte obiecte participă în relație (ex. de multiplicitate: nespecificată, exact una, zero sau mai multe (multe, nelimitat), una sau mai multe, zero sau una, etc.)

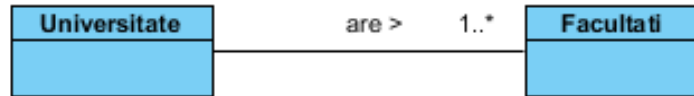
- Numărul de instanțe ale unei clase în raport cu o instanță a celeilalte clase
- Specificat pentru fiecare capăt al asocierii

Asocierile sunt implicit bidirecționale, dar adesea este de dorit să se restrângă navigarea la o singură direcție

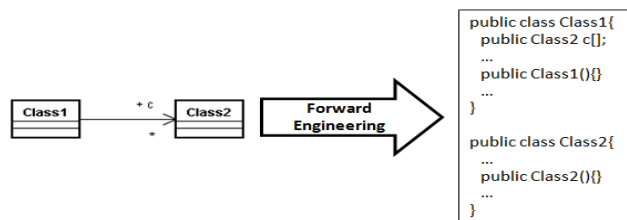
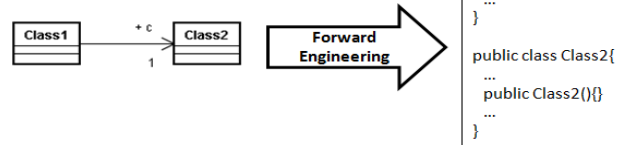
- Dacă navigarea este restricționată, se adaugă o săgeată pentru a indica direcția de navigare
- **Exemplu de relatie de asociere de tip one – to – one intre doua clase. O Universitate este administrată de un singur Rector, iar un Rector administrează o singură Universitate**



- **Exemplu de relatie de asociere de tip one – to – many intre doua clase. O Universitate are mai multe Facultăți.**

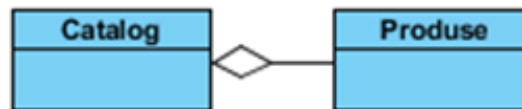


Cum se transcrie o relatie de asociere la nivel de cod:



c) Exemplu de relatie de agregare între două clase.

Catalogul poate avea mai multe **Produse** în același timp, un **Produs** poate exista chiar dacă acel **Catalog** nu există.



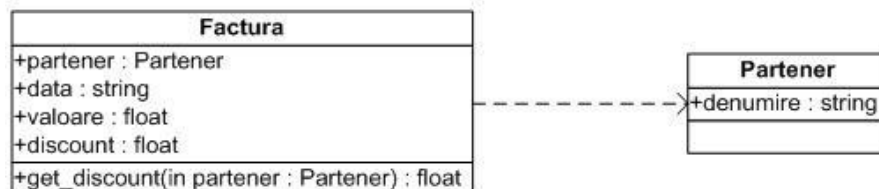
d) Exemplu de relatie de compoziție între două clase.

Instanța clasei **Comisie** există atâta timp cât există o instanță din clasa **Examen**.

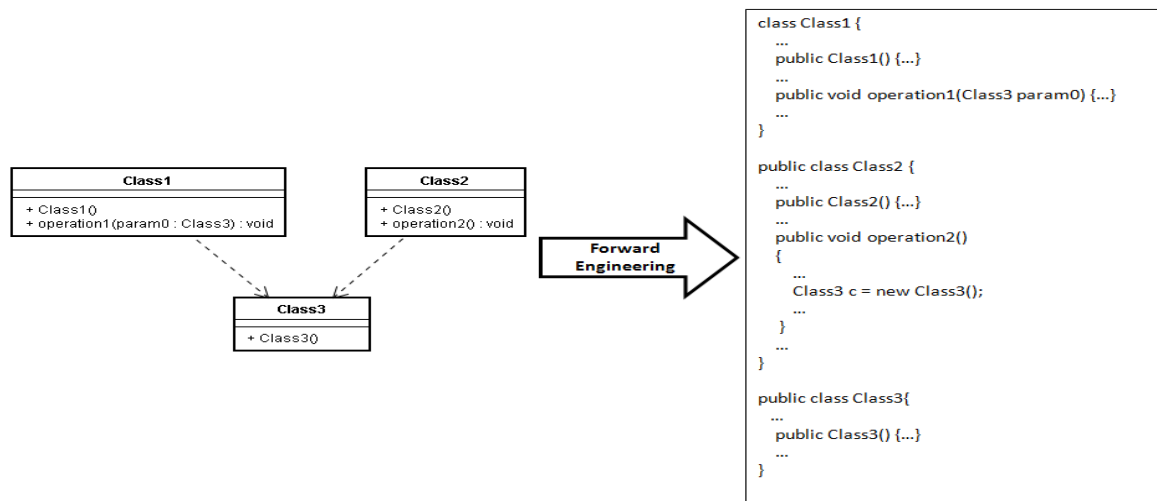


e) Exemplu de relatie de dependentă între două clase.

Clasa **Factura** are o metoda ca primește ca parametru un obiect instant al clasei **Partener**.

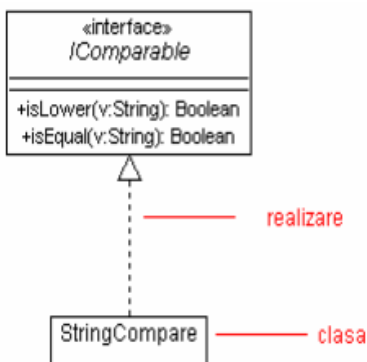


Cum se transcrie o relatie de dependenta la nivel de cod:

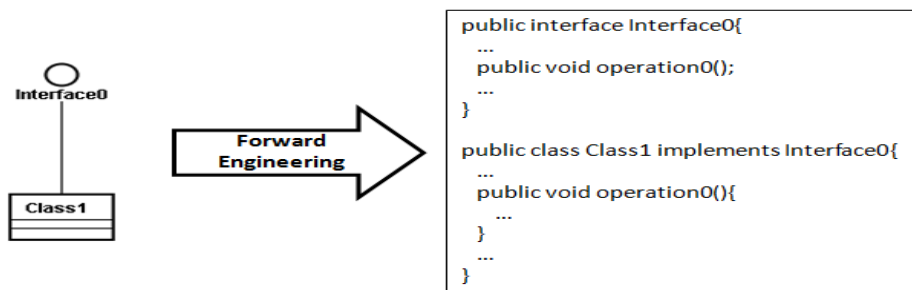


f) **Exemplu de relatie de realizare.**

Interfata **IComparable** este implementata de clasa **StringCompare**.



Cum se transcrie o relatie de realizare la nivel de cod:



3. **Exerciti care for fi implementate in cadrul laboratorului:**

1. Definiți clasa **Form** cu un membru **color** de tip **String**, o metoda **getArea()** care pentru început va intoarce 0 și o metodă **toString()** care va returna această culoare. Clasa va avea, de asemenea: (i) un constructor fără parametri si (ii) un constructor ce va inițializa culoarea. Din ea derivați clasele **Triangle** și **Circle**. Clasa **Triangle** va avea 2 membri **height** si **base** de tip **float**. Clasa **Circle** va avea membrul **radius** de tip **float**.

- Clasele vor avea: constructori fără parametri, constructori care permit inițializarea membrilor. Identificați o modalitate de **reutilizare** a codului existent.
- Instanțiați clasele **Triangle** și **Circle** și apelați metoda **toString()** pentru fiecare instanță.
- suprascrieti metoda **getArea()** pentru a întoarce aria specifică figurii geometrice.
- Adăugați metode **toString()** în cele două clase derivate, care să returneze tipul obiectului, culoarea și aria. De exemplu: pentru clasa **Triangle**, se va afișa: "Triunghi: roșu 10", pentru clasa **Circle**, se va afișa: "Cerc: verde 12.56"
- Modificați implementarea **toString()** din clasele derivate astfel încât aceasta să utilizeze implementarea metodei **toString()** din clasa de bază.
- Adăugați o metodă **equals** în clasa **Triangle**. Justificați criteriul de echivalență ales.
- Upcasting. Creați un vector de obiecte **Form** și populați-l cu obiecte de tip **Triangle** și **Circle** (upcasting). Parcurgeți acest vector și apelați metoda **toString()** pentru elementele sale. Ce observați?
- Downcasting. Adăugați clasei **Triangle** metoda **printTriangleDimensions** și clasei **Circle** metoda **printCircleDimensions**. Implementarea metodelor constă în afișarea bazei și înălțimii respectiv razei.
- Parcurgeți vectorul de la exercițiul anterior și, folosind downcasting la clasa corespunzătoare, apelați metodele specifice fiecărei clase (**printTriangleDimensions** pentru **Triangle** și **printCircleDimensions** pentru **Circle**). Pentru a stabili tipul obiectului curent folosiți operatorul **instanceof**.

2. Implementați o aplicație care constă din următoarele clase: **Person**, **Teacher**, **Student**, **Course**, **Departament**, **School**. Clasa **Course** are ca atribute numele cursului de tip **String** și **anul** la care se predă acest curs de tip **integer**. Clasa **Person** are ca și atribute **numele** și **adresa** persoanei. Clasa definește constructori, metode **get** și **set**, suprascrie metoda **toString** și **equals** din clasa **Object**. Clasa **Person** este extinsă de către clasele **Student** și **Teacher**. Clasa **Student** are în plus atributul **nrMatricol** de tip **long**, iar clasa **Teacher** are în plus atributul **profesorID** de tip **String**. Ambele clase specializează metodele **toString** și **equals** moștenite din superclasa. Un student poate participa la mai multe cursuri, iar un profesor predă un singur curs. Clasa **Departament** are ca și atribut un nume de tip **String** și metode pt adăugarea unui nou profesor, pt ștergerea unui profesor, pt căutarea unui profesor după nume, pentru afișarea tuturor profesorilor care predau la un anumit an. Dintr-un departament pot face parte unul sau mai mulți profesori. Clasa **School** are ca și atribute numele și adresa școlii de tip **String** și definește următoarele metode: **addStudent**, **removeStudent**, **getStudent**, **addDepartment**, **removeDepartment**, **getDepartment**, **listDepartments**, **listStudents**, etc.. La o școală există mai multe departamente și pot învăța mai mulți studenți. În diagrama de mai jos este reprezentată structura claselor și relațiile dintre ele.

