Curs 3: Extinderea claselor

Cuprins

1. Reminder - Mostenire = derivare = extindere

Mostenire/

derivare/

extindere

-> reprezinta posibilitatea de a defini o clasa care extinde o alta clasa deja existenta, preluând funcționalitățile sale și adăugând altele noi

-> oferă posibilitatea de a dezvolta pas cu pas o aplicație, procedeu care poarta numele de **dezvoltare incrementală (incremental development):**

- se poate utiliza un cod deja funcțional, respectiv testat, și adăuga un alt cod nou la acesta, în felul acesta izolându-se eventualele erori din codul nou adăugat.

-> permite **polimorfismul dinamic prin suprascriere** (subclasa rescrie metoda superclasei)

-> modeleaza relatia **IS-A** (spre deosebire de relatia HAS-A)

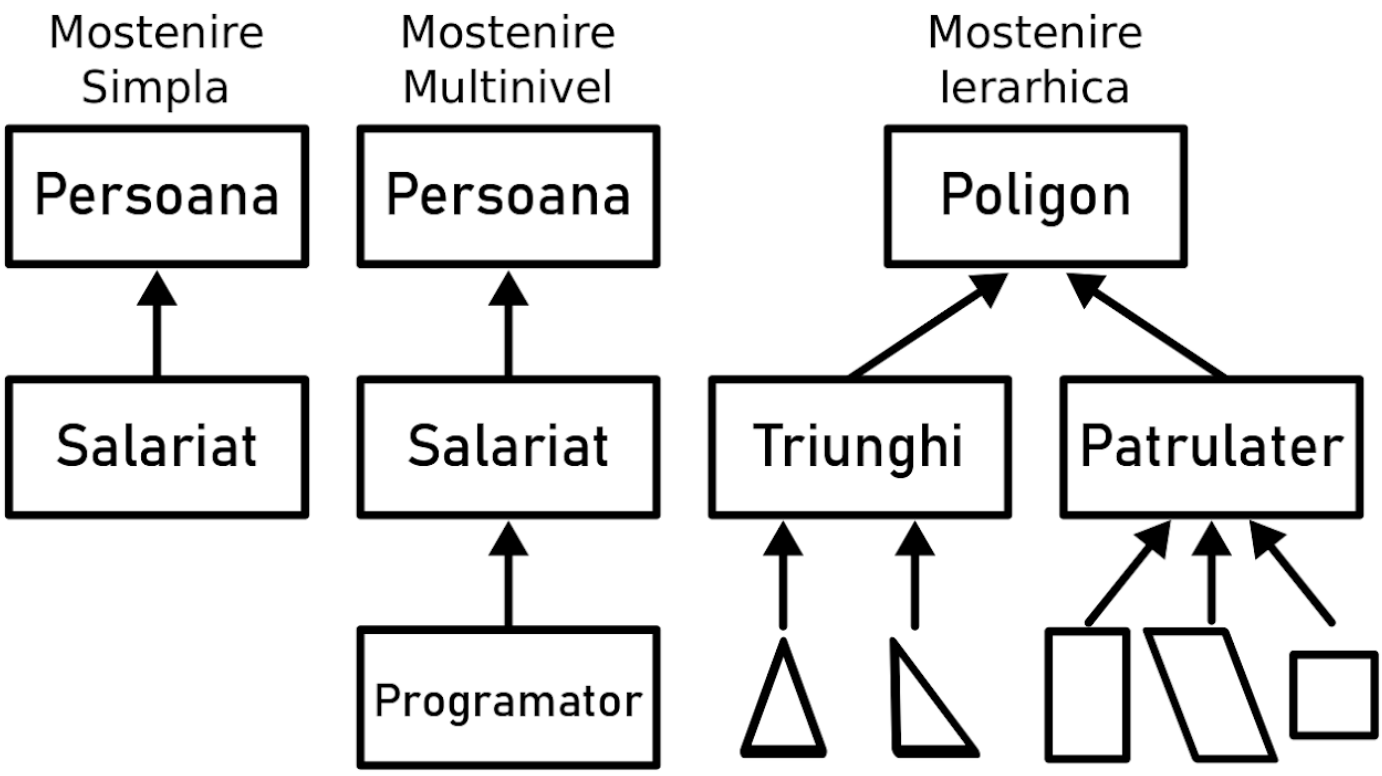
Terminologie: Clasa care se extinde se numește **superclasă**, iar cea care preia datele și funcționalitățile se numește **subclasă**.

```
class Subclasa extends Superclasa {

    date și metode membre extra

}
```

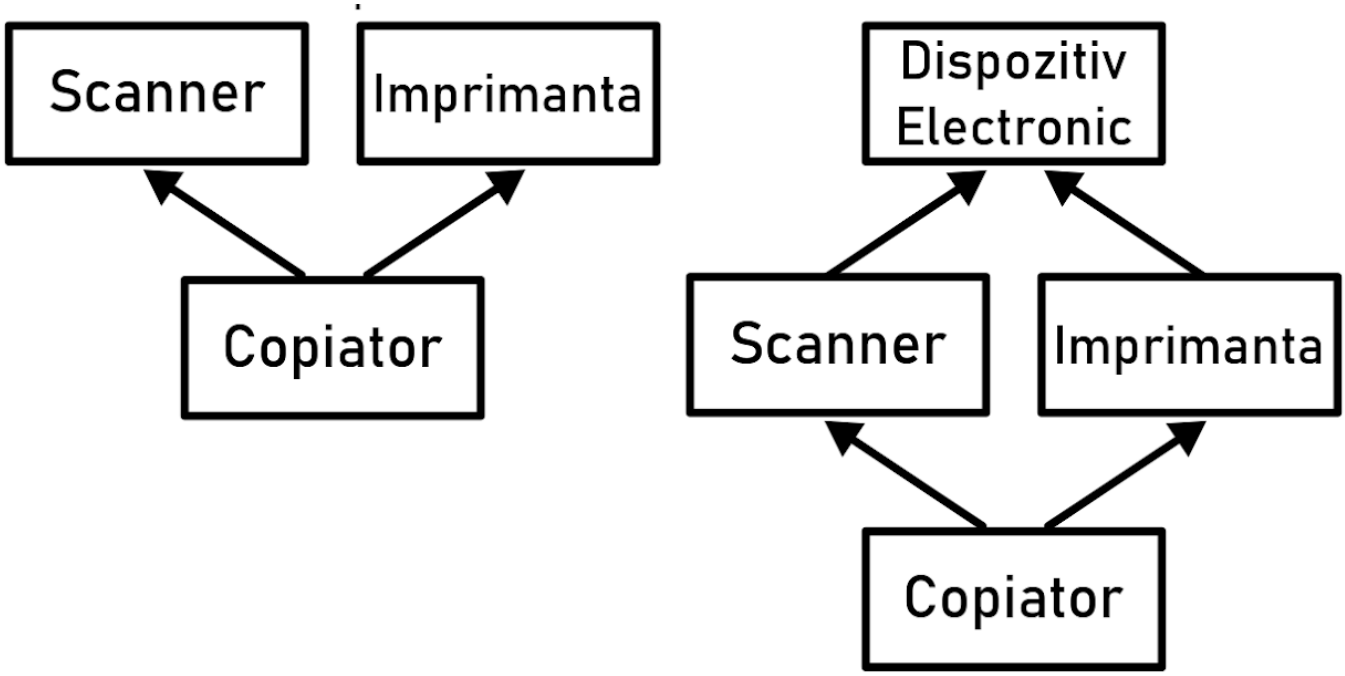
2. Tipuri de mostenire



```
    int id;
    String nume;          ...}
public class Salariat extends Persoana {
    float salariu;        ...}
public class Programator extends Salariat {
    int idProiectCurent;      ...}

public class Poligon{
    int nrLaturi;
    List<SegmentGeometric> listaSegmente;
    float aria(){...}; //convex??
    float perimetru(){...};
    ...
}

public class Triunghi extends Poligon {
    float unghi1, unghi2, unghi3;
    float aria(){...};
    ...
}
public class Patrulater extends Poligon {
    SegmentGeometric diagonala1, diagonala2;
    float aria(){...};
    ...
}
```



In Java **nu exista mostenire multipla pentru clase**

(folosind class **extends** class1~~,class2~~)

Dar mecanismul exista sub forma de **interfete**

(folosind class **implements** interface1, interface2)

3. Caracteristicile mostenirii

- orice clasa din Java extinde clasa **Object** (este superclasa tuturor claselor)
- moștenirea este întotdeauna **publică** și **singulară**,
    1. respectiv membrii din superclasă își **păstrează modificatorul de acces**,
    2. iar orice clasă din ierarhia de clase are **o singură superclasă directă**

Ce se moștenește?

• Membrii privați nu sunt moșteniți, dar pot fi accesați prin metode publice sau protejate din superclasă.

• Nu se moștenesc datele membre și metodele statice.

• Metodele constructor, nefiind considerate metode membre ale unei clase, nu se moștenesc, dar un constructor din subclasă poate apela constructorii din superclasa, folosind expresia super([argumente]).

• Cuvântul cheie **super** poate fi folosit și pentru a accesa date membre și metode din superclasă, astfel: **super**.metoda(lista arg) sau **super**.dată_membră.

! Din orice clasa ne putem referi la **super** pentru ca implicit extinde **Object** !

4. Constructorul superclasei

Observatii

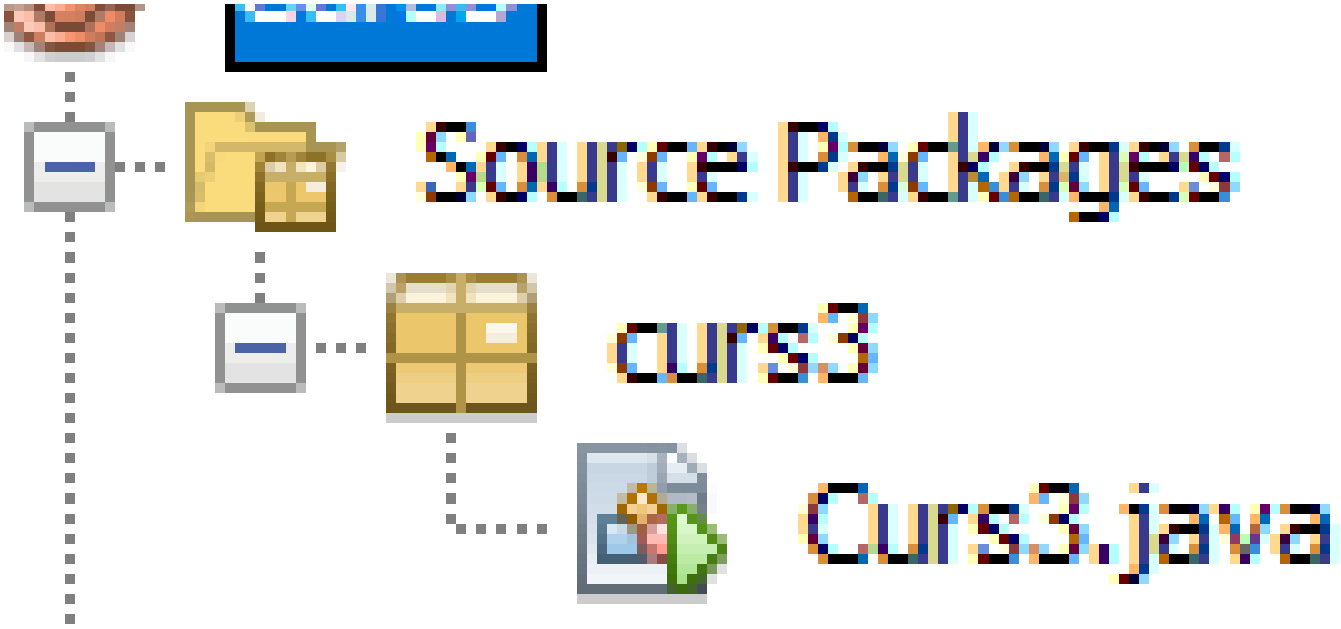
• La instanțierea unui obiect de tip subclasă se apelează constructorii din ambele clase, mai întâi cel din superclasă și apoi cel din subclasă!

• Apelul **constructorului superclasei**, dacă există, trebuie să fie prima instrucțiune din constructorul subclasei (un obiect al subclasei este mai întâi de tipul superclasei).

• Dacă nu se introduce în subclasă apelul explicit al unui constructor al superclasei, atunci compilatorul va încerca să apeleze **constructorul fără argumente al superclasei**. Aceasta poate cauza o eroare în cazul în care superclasa nu are definit un constructor fără argumente, dar are definit unul cu argumente!

• Constructorul cu argumente din subclasă conține, de regulă, argumente pentru inițializarea datelor membre din superclasă.

5. Exemplu super constructor


Source Packages
curs3
Curs3.java

```
    String nume;
    int varsta;
    public Persoana(String nume, int varsta) {
        this.nume = nume;
        this.varsta = varsta;
    }
    @Override
    public String toString() {
        return nume + " " + varsta;
    }
    //metode de tip set/get pentru datele membre nume și vârsta
}

class Student extends Persoana {
    String facultate;
    int grupa;
    double medie;
    public Student(String nume, int varsta, String facultate, int grupa, double medie) {
        super(nume, varsta); //trebuie sa fie la inceput
        this.facultate = facultate;
        this.grupa = grupa;
        this.medie = medie;
    }
    @Override
    public String toString() {
        return super.toString() + " " + facultate + " " + grupa + " " + medie;
    }
    //metode de tip set/get pentru datele membre facultate, medie și grupă
}

public class Curs3 {
    public static void main(String[] args) {
        Persoana persoana = new Persoana("Ion", 20);
        System.out.println(persoana);
        Student student = new Student("Dan",19,"Informatica",143,9);
        System.out.println(student);
        Persoana persoanaStudent = new Student("Ana",21,"Informatica",332,9.5);
        System.out.println(persoanaStudent);
    }
}
```

6. Suprascriere / Overriding / Polimorfism dinamic

= concept puternic în limbajele orientate pe obiecte

= permite subclasei să redefinească o metoda moștenită și să-i modifice comportamentul

-> la executare, în raport cu tipul obiectului se va invoca metoda corespunzătoare.

-> metoda din superclasă este „ascunsă” de cea redefinită în subclasa sa.

Caracteristici

• **Tipul intors este acelasi sau poate fi mai specific** (adica mai jos in ierarhia de clase).

◦ void -> void

◦ tip de date primitiv -> același tip de date primitiv

◦ referință de tip superclasă -> referință de tip superclasă **sau de tip subclasă**

• Nu se pot redefini metodele de tip **final**.

• Se poate utiliza adnotarea **@Override** pentru a-i indica explicit compilatorului faptul că se va redefini o metodă din superclasă. Astfel, dacă metoda respectivă nu există în superclasă, compilatorul va furniza o eroare.

• (**Variable shadowing**) O dată membră din superclasă va fi ascunsă prin redeclararea sa în subclasă, dar poate fi totuși accesată prin **super**.dată_membră.

• O metodă din superclasă poate fi redefinită în subclasă, dar poate fi totuși accesată prin **super**.metodă([parametrii]).

• Metodele **statice** pot fi redefinite doar prin metode **statice** (dar le ascund!!!), iar cele **nestatice** doar prin metode **nestatice**.

• Pentru o metodă redefinită se poate schimba **modificatorul de acces, dar fără ca nivelul de acces să scadă**.

7. Exemplu Suprascriere / Overriding

```
    public Angajat(String nume, int varsta, String firma, String functie, double salariu) {
        super(nume, varsta);
        this.firma = firma;
        this.functie = functie;
        this.salariu = salariu;
    }
    @Override
    public String toString() {
        return super.toString() + " " + firma + " " + functie + " " + salariu;
    }
    double calculeazaVenit() {
        return salariu;
    }
}

class Economist extends Angajat {
    int treapta_profesionala;//treapta profesională -> număr natural cuprins între 0 și 5
    //sporurile procentuale corespunzătoare treptelor profesionale
    static final double sporuri[] = {5, 10, 15, 20, 25, 30};
    public Economist(int treapta_profesionala, String nume, int varsta,
                     String firma, String functie, double salariu) {
    super(nume, varsta, firma, functie, salariu);
    this.treapta_profesionala = treapta_profesionala;
}
    @Override
    double calculeazaVenit() {
        return salariu + salariu * sporuri[treapta_profesionala] / 100;
    }
    @Override
    public String toString() {
        return super.toString() + " " + treapta_profesionala;
    }
}
```

8. Diferentele dintre tipurile de polimorfism

se poate realiza și doar în cadrul unei singure clase	se poate realiza doar într-o subclasă a unei superclase
la compilare (legare statică)	la rulare (legare dinamică)
mai rapidă	mai lentă
metodele de tip final sau private pot fi supraîncărcate	metodele de tip final nu pot fi rescrise
nivelul de acces nu contează	nivelul de acces nu trebuie să fie mai restrictiv decât cel al metodei din superclasă
tipul de date returnat nu contează	tipul de date returnat trebuie să respecte principiul de covarianță
lista parametrilor formali trebuie să fie diferită	lista parametrilor formali trebuie să fie identică

9. Nivelul de acces al datelor/metodelor membre la mostenire

Modifier	Class	Package	Subclass	Global
Public	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	No
Default	Yes	Yes	No	No
Private	Yes	No	No	No

10. Clasa Object

- Fiecare clasă Java este un descendent al clasei **Object**, astfel încât orice clasă moștenește metodele clasei Object.
- Aceste metode nu se utilizează în orice aplicație, dar, dacă sunt folosite, atunci trebuie cunoscute câteva principii de redefinire a lor.

public final Class getClass()

▪ Este o metodă de tip final (nu mai poate fi redefinită) care returnează un obiect de tip Class care conține detalii despre clasa instanțiată în momentul executării programului.

▪ Clasa **Class** este definita în **java.lang**, nu are constructor public, astfel încât obiectul este construit implicit de către mașina virtuală Java cu ajutorul unor metode de tip factory.

public String toString()

▪ Metoda returnează o reprezentare a obiectului sub forma unui obiect de tip **String**.

▪ Trebuie redefinită în fiecare clasă, deoarece acesta reprezentare este dependentă de fiecare obiect. De regulă, se construiește un șir de caractere care conține valorile câmpurilor.

▪ Implicit metoda **toString** afișează un șir format astfel:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

boolean equals(Object obj)

▪ În limbajul Java, două obiecte se pot compara în două moduri, folosind:

operatorul ==

care verifică dacă două obiecte **sunt egale din punct de vedere al referințelor**, adică dacă au aceeași referință, însă **nu se testează echivalența lor din punct de vedere al conținutului**!

```
Persoana p1 = new Persoana("Matei", 23);
Persoana p2 = p1;
System.out.println(p1==p2);//se va afișa true
Persoana p3 = new Persona("Matei", 23);
System.out.println (p1==p3);//se va afișa false
```

care, implicit, verifică dacă două obiecte au aceeași referință, folosind **operatorul ==**, după cum se poate observa din exemplul următor:

System.out.println (p1.**equals**(p3));//se va afișa false

De cele mai multe ori, în aplicații este necesară o comparare a două obiecte și din punct de vedere al conținutului, caz în care trebuie redefinită metoda **equals**().

Exemplu: Redefinirea metodei equals pentru clasa Persoana cu datele membre nume și varsta:

```
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
         return false;
    if (getClass() != obj.getClass())
        return false;
    //e un obiect de tip Persoana !
    Persoana pers = (Persoana) obj;
    if (this.varsta != pers.varsta)
        return false;
    if (!Objects.equals(this.nume, pers.nume))
        return false;
    return true;
}
```

• Se poate observa faptul că pentru a compara numele celor două persoane se preferă utilizarea metodei **equals**() din clasa utilitară **Objects** (introdusă din versiunea 1.7), deoarece metodele **equals**() și **compareTo**() din clasa **String** pot furniza excepții în cazul în care argumentele sunt de tip **null**. În schimb, metoda equals() din clasa Objects va compara cele două șiruri astfel:

◦ dacă ambele obiecte String sunt de tip **null**, atunci metoda va returna valoarea **true**;

◦ dacă un un obiect String este **null**, iar celălalt nu este, va returna **false**;

◦ dacă ambele obiecte de tip String nu sunt null, va returna rezultatul obținut prin apelarea metodei **equals**() din clasa String.

public int hashCode()

În Java, codul hash al unui obiect este un număr întreg care este dependent de conținutul său.

Implicit, codul hash este calculat de către mașina virtuală Java, utilizând un algoritm specific care nu utilizează valorile câmpurilor obiectului respectiv.

Astfel, pentru a se evidenția faptul că un obiect își modifică starea pe parcursul executării programului sau pentru a ne asigura de faptul că două obiecte egale din punct de vedere al conținutului au același cod hash, se redefinește metoda **hashCode**(), astfel încât algoritmul de hash **să utilizeze concret starea obiectului** (valorile datelor membre).

În acest sens, clasa Objects conține o metodă cu număr variabil de argumente care calculează codul hash pentru un anumit obiect folosind valorile câmpurilor sale: **Objects.hash(câmp_1, câmp_2, ...)**

```
@Override
public int hashCode() {
    return Objects.hash(nume, varsta);
}
```

Mai multe detalii referitoare la modalități de implementare a metodei **hashCode**() puteți găsi aici: https://www.baeldung.com/java-hashcode.

Observație: Implementările metodelor **equals**() și **hashCode**() trebuie să țină cont de următoarele reguli:

• metoda **hashCode**() trebuie să returneze aceeași valoare în timpul rulării unei aplicații, indiferent de câte ori este apelată, dacă starea obiectului nu s-a modificat, dar nu trebuie neapărat să furnizeze aceeași valoare în cazul unor rulări diferite;

• două obiecte egale din punct de vedere al metodei **equals**() trebuie să fie egale și din punct de vedere al metodei **hashCode**(), deci trebuie să aibă și hash code-uri egale;

• două obiecte diferite din punct de vedere al conținutului nu trebuie neapărat să aibă hash-code-uri diferite, dar, dacă acest lucru este posibil, se vor obține performanțe mai bune pentru operațiile asociate unei tabele de dispersie.

11. Polimorfism

Conceptul în sine este o consecință a faptului ca orice obiect din Java poate fi referit prin tipul său sau prin tipul unei superclase, respectiv se poate realiza, într-un mod implicit, conversia unei subclase la o superclasă a sa mecanism care poartă denumirea de upcasting.

Upcasting

Considerăm clasa A extinsă de clasa B. Pot avea loc următoarele instanțieri:

```
B b = new B(); //referirea obiectului printr-o referință de tipul clasei
A a = new B(); //referirea obiectului printr-o referință de tipul superclasei
```

Observație: În cazul celei de-a doua instanțieri, nu este convertită valoarea, ci referința obiectului!!! Spunem că obiectul a are tipul declarat A și tipul real B!

Downcasting

În sens invers, downcasting-ul reprezintă accesarea unui obiect de tipul superclasei folosind o referință de tipul unei subclase și nu este implicit, necesitând o conversie explicită!

Exemplu: Considerăm clasa Angajat și două subclase ale sale, Economist și Inginer.

```
//( ) ct, deoarece upcasting-ul se realizează implicit
```

```
//eroare la compilare, deoarece downcasting-ul nu se realizează implicit
Inginer p = b;
//fără eroare la compilare, deoarece downcasting-ul a fost realizat explicit
Inginer p = (Inginer)b;
```

Totuși, în momentul executării ultimei instanțieri de mai sus, se va declanșa excepția ClassCastException și rularea programului se va termina. Pentru a preveni acest lucru, se verifică în prealabil dacă se poate efectua downcasting-ul respectiv, astfel:

Instanceof

```
Inginer p = null;
if(b instanceof Inginer)
    p = (Inginer)b;
```

Exemplu metoda statica

Considerăm o clasă A care este extinsă de o clasă B, astfel:

```
class A{
    int dată_membră = 3;
    void metoda1() {
        System.out.println("Metoda 1 din clasa A!");
    }
    static void metoda2(){
        System.out.println("Metoda statică 2 din clasa A!");
    }
}
class B extends A {
    int dată_membră = 4;
    void metoda1() {
        System.out.println("Metoda 1 din clasa B!");
    }
    static void metoda2(){
        System.out.println("Metoda statică 2 din clasa B!");
```

Se poate observa cum subclasa B redefinește atât data membră, cât și cele două metode.

Rulând secvența de instrucțiuni

```
A ob = new B();    //polimorfism
System.out.println("Data membră = " + ob.dată_membră);
ob.metoda1();
ob.metoda2();
```

se va afișa:

Data membră = 3

Metoda 1 din clasa B!

Metoda statică 2 din clasa A!

Analizând exemplul de mai sus, observăm următoarele aspecte:

• ob.dată_membră este 3, deoarece câmpurile se accesează după tipul declarat, ci nu după tipul real;

• ob.metoda1() utilizează implementarea din subclasă, respectiv metoda redefinită, deoarece selecția se realizează după tipul real;

• ob.metoda2 () utilizează implementarea din superclasă, deoarece metodele statice nu se redefinesc, deci selecția se realizează după tipul declarat.

În concluzie, o metodă de instanță este invocată în raport de tipul real, tip care se identifică la executare (runtime). Conceptul se mai numește și legare dinamică sau legare târzie (late binding).

Late binding

Exemplu:

```
Scanner sc = new Scanner(System.in);
double d = sc.nextDouble();
A Ob;
if (d > 0) Ob = new A();
else Ob = new B();
```

12. Clase abstracte

În momentul în care dorim să abstractizăm un anumit concept/fenomen din lumea reală, este posibil să nu-i putem defini comportamentul său în orice situație. De exemplu, despre orice angajat, indiferent de domeniul său de activitate, trebuie să cunoaștem anumite informații precum nume, adresă, firma la care este angajat, numărul de ore lucrate zilnic etc. De asemenea, pentru orice angajat trebuie să putem calcula numărul total de ore lucrate de acesta într-o anumită perioadă de timp, să-i determinăm vechimea totală etc. Totuși, modalitatea de calcul a salariului unui angajat depinde de profesia sa (inginer, profesor, medic etc.), de anumite sporuri specifice (are anumite deduceri de impozit, lucrează noaptea sau nu, lucrează în week-end-uri, lucrează în mediu toxic sau nu etc.). În acest caz, o metodă care să calculeze salariul unui angajat fie va lua în considerare toate cazurile posibile, ceea ce conduce la o eficiență scăzută, fie va fi declarată abstractă, urmând a fi implementată obligatoriu în clase specializate (Profesor, Inginer, Medic etc.), care vor extinde clasa Angajat.

Exemplu:

```
public abstract class Angajat {
    double salariu_baza;
    ......
    abstract public double calculSalariu();
}

class Paznic extends Angajat{
     ......
     static double spor_de_noapte = 0.25;
     ......
     Public double calculSalariu() {
          return salariu_baza + salariu_baza * spor_de_noapte;
     }
}
```

Observații:

• O clasă abstractă nu se poate instanția, deoarece nu se cunoaște integral funcționalitatea sa.

• O clasă abstractă poate să conțină date membre de instanță, constructori și metode publice, astfel încât subclasele sale pot apela constructorul din superclasă, respectiv pot redefini membrii săi.

Polimorfismul se poate implementa cu ușurință folosind clase abstracte. În exemplul de mai sus, clasa Angajat este o clasă abstractă, deci nu poate fi instanțiată. Astfel, în momentul compilării este posibil să nu se cunoască tipul concret al unui angajat, dar un obiect de tip subclasă (Inginer, Profesor etc.) poate fi accesat printr-o referință de tipul superclasei, respectiv prin referința clasei abstracte (polimorfism!).

Exemplu: Considerăm clasa abstractă Angajat, care conține metoda abstractă double calculSalariu() și 3 subclase ale sale Inginer, Profesor și Economist, toate trei implementând în mod specific metoda calculSalariu()și având definiți constructori și alte metode necesare. Fișierul text angajați.csv conține pe fiecare linie informații despre fiecare angajat al unei firme, despărțite între ele prin virgule, iar fiecare linie începe cu un șir de caractere care indică profesia angajatului respectiv. Pentru a încărca într-un singur tablou unidimensional informațiile despre toți angajații firmei, indiferent de profesie, vom declara un tablou a cu elemente de tipul clasei abstracte Angajat (de fapt, referințe!) și, folosind polimorfismul, vom instanția pentru fiecare element al tabloului un obiect de tipul uneia dintre cele 3 subclase:

```
.....
Scanner in = new Scanner(new File("exemplu.txt"));
int n = in.nextInt();
in.nextLine();
Angajat []a = new Angajat[n];
String linie;
for(int i = 0; i < n; i++){
    linie = in.nextLine();
    String []aux = linie.split(",");
    switch(aux[0].toUpperCase()){
        //se apelează constructorul corespunzător fiecărei subclase
        case "PROFESOR":
            a[i] = new Profesor(...);
            break;
```

```
            a[i] = new Inginer(...);
            break;
        case "ECONOMIST":
            a[i] = new Economist(...);
        break;
    }
}
for(int i = 0; i < n; i++)
    System.out.println(a[i].getNume() + " -> " + a[i].calculSalariu() + " RON");
.....
```

Datorită polimorfismului, pentru fiecare element al tabloului a se va apela varianta metodei calculSalariu() din subclasa sa!