

EXTINDEREA CLASELOR

Numita și **derivare, moștenirea** este un mecanism de refolosire a codului, specific limbajelor orientate obiect și reprezintă posibilitatea de a defini o clasă care extinde o alta clasă deja existentă, preluând funcționalitățile sale și adăugând altele noi. Pe lângă reutilizarea codului, moștenirea oferă posibilitatea de a dezvolta pas cu pas o aplicație, procedeu care poartă numele de *dezvoltare incrementală (incremental development)*. Astfel, 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.

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

Sintaxa:

```
class Subclasa extends Superclasa
{
    date și metode membre noi
}
```

Moștenirea definește o relație între superclasă și subclasa să de tip IS_A și conduce definirea unei ierarhii de clase care are rădăcina în clasa **Object**. Practic, orice clasa din Java extinde clasa **Object**, denumită *superclasa claselor*. Din sintaxa de mai sus se poate observa faptul că, în limbajul Java, moștenirea este întotdeauna *publică* și *singulară*, respectiv membrii din superclasă își păstrează modificatorul de acces, iar orice clasă din ierarhia de clase are o singură superclasă directă.

Ce se moștenește?

- Subclasă moștenește toți membrii publici, protejați și implicați din superclasă.
- Membrii privați nu sunt moșteniți, dar pot fi accesati prin metode publice sau protejate din superclasă.
- 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`.

Observații:

- 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.

Exemplu:

```

class Persoana
{
    private String nume;
    private int varsta;

    public Persoana(String nume , int varsta)
    {
        this.nume = nume;
        this.varsta = varsta;
    }

    @Override
    protected String toString() { return nume + " " + varsta; }

    //metode de tip set/get pentru datele membre nume și vârstă
}

class Student extends Persoana
{
    private String facultate;
    private int grupa;
    private double medie;

    public Student(String nume, int varsta, String facultate, int grupa, double medie)
    {
        super(nume, varsta);
        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
}

```

Mecanismul de redefinire a unei metode (overriding)

Mecanismul de redefinire reprezintă un concept puternic în limbajele orientate pe obiecte care permite subclasei să redefinească o metoda moștenită și să-i modifice comportamentul. Astfel, la executare, în raport cu tipul obiectului se va invoca metoda corespunzătoare. Practic, metoda din superclasă este „ascunsă” de cea redefinită în subclasă sa.

Observații:

- O metodă din subclasă care redefinește o metodă din superclasă trebuie să păstreze lista inițială a parametrilor formali.

- 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 superclasa. Astfel, dacă metoda respectivă nu există în superclasa, compilatorul va furniza o eroare.
- O dată membră din superclasa va fi ascunsă prin redeclararea sa în subclasa, dar poate fi totuși accesată prin **super.dată_membră**.
- O metodă din superclasa poate fi redefinită în subclasa, dar poate fi totuși accesată prin **super.metodă([parametrii])**.
- Pentru o metodă redefinită se poate schimba modificadorul de acces, dar fără ca nivelul de acces să scadă.
- Tipul returnat de o metodă din subclasa care redefinește o metodă din superclasa trebuie să fie unul covariant tipului de date inițial, respectându-se astfel *principiul de covarianță* (sau *principiul de substituție Liskov*):
 - void ↔ void
 - tip de date primitiv ↔ același tip de date primitiv
 - referință de tip superclasa ↔ referință de tip superclasa sau de tip subclasa

Exemplu:

```
class Angajat extends Persoana
{
    protected String firma , functie;
    protected double salariu;

    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;
    }

    public double calculeazaVenit()
    {
        return salariu;
    }
}

class Economist extends Angajat
```

```
{
    //treapta profesională -> număr natural cuprins între 0 și 5
    private int treapta_profesionala;
    //sporurile procentuale corespunzătoare treptelor profesionale
    private 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
    public double calculeazaVenit() {
        return salariu + salariu*sporuri[treapta_profesionala]/100;
    }

    @Override
    public String toString() {
        return super.toString() + " " + treapta_profesionala;
    }
}
```

Mecanismul de redefinire, alături de cel de supraîncărcare, permite definirea polimorfismului. Totuși, între cele două concepte există diferențe clare:

SUPRAÎNCĂRCARE (OVERLOADING)	REDEFINIRE (OVERRIDING)
• 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

Clasa Object

Clasa `Object`, definită în pachetul `java.lang`, definește și implementează un comportament comun pentru orice obiect Java. Fiecare clasă Java este un descendant 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 definită 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 sir de caractere care conține valorile câmpurilor.
 - Implicit metoda `toString` afișează un sir 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
```

- **metoda `boolean equals()`** 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;

        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 siruri astfel:
 - dacă ambele obiecte `String` sunt de tip `null`, atunci metoda va returna valoarea `true`;
 - dacă unul dintre obiecte `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 redefineste 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(camp_1, camp_2, ...)`.

Exemplu: Calculul codului hash pentru un obiect de tip `Persoana`

```

@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.

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 superclăsă a sa mecanism care poartă denumirea de *upcasting*.

Considerăm clasa B extinsă de clasa A. 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!*

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`.

```
//corect, deoarece upcasting-ul se realizează implicit
Angajat a = new Economist();
Angajat b = new Inginer();

//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:

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

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(); //de instantă
ob.metoda2(); //statică
```

se va afișa:

```
Data membră = 3 //după tipul declarat
Metoda 1 din clasa B!//după tipul real
Metoda statică 2 din clasa A! //după tipul declarat
```

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).

Exemplu:

```
Scanner sc = new Scanner(System.in);
double d = sc.nextDouble();
```

A Ob;//de tipul superclasa

```
if (d>0) Ob = new A(...); //de tipul concret ->tipul subclasa
else Ob = new B(...);

ob.met(...);           //se apelează după tipul real identificat la executare
```

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 **obiectiv** 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.
- Dacă o subclasă a unei clase abstracte nu oferă implementări pentru toate metodele abstracte moștenite, atunci subclasa este, de asemenea, abstractă, deci nu poate fi instanțiată.
- O clasă abstractă poate să conțină date membre de instanță, constructori și metode publice, astfel încât subclasele sale pot apela constructorul din superclasa, 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 definiții 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 sir 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;
        case "INGINER":
            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");
.....

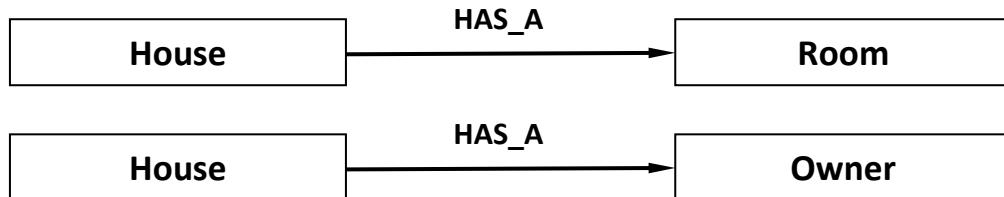
```

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

AGREGARE ȘI COMPOZIȚIE

- Agregarea și compozitia reprezintă alte două modalități de interconectare (asociere) a două clase, alături de mecanismul de extindere a claselor (moștenire).
- Practic, agregarea și compozitia reprezintă alte modalități de reutilizare a codului.
- Asocierea a două clase se realizează prin încapsularea în clasa container a unei referințe, de un tip diferit, către un obiect al clasei asociate (încapsulate).
- Conceptual, compozitia este diferită de agregare în raport de ciclul de viață al obiectului încapsulat, astfel:
 - dacă ciclul de viață al obiectului încapsulat este dependent de ciclul de viață al obiectului container, atunci relația de asociere este de tip *compoziție* (strong association);
 - dacă obiectul încapsulat poate să existe și după distrugerea containerului său, atunci relația de asociere este de tip *agregare* (weak association).

Exemple:



- Se poate observa cu ușurință faptul că relația de asociere dintre clasele `House` și `Room` este una de tip compoziție (dacă este distrusă întreaga casă, atunci, în mod automat, va fi distrusă și camera respectivă), iar relația de asociere dintre clasele `House` și `Owner` este una de tip agregare (chiar dacă este distrusă întreaga casă, proprietarul său poate să trăiască în continuare 😊)).
- Din punct de vedere al implementării, diferențierea dintre cele două tipuri de asocieri se realizează prin modul în care obiectul container încapsulează referința spre obiectul asociat. Astfel, în cazul unei relații de agregare este suficientă încapsularea în clasa container a unei referințe spre obiectul asociat, deoarece acesta poate exista și independent:

<pre> class Person{ private String name; private String SSN; </pre>	<pre> class House{ private String address; private Person owner; </pre>
---	---

```

} .....      .....  

           public House(Person owner,...) {  

           this.owner = owner;  

           .....  

           }  

}

```

În cazul unei relații de compoziție se va încapsula în clasa container o referință a unei copii locale a obiectului asociat:

<pre> class Room{ private float width; private float length; public Room(Room r){ this.width = r.width; } } </pre>	<pre> class House{ private String address; private Room dining; public House(Room dining,...){ this.dining = new Room(dining); } } </pre>
--	---

- În exemplul de mai sus, am presupus faptul că în clasa `Room` este definit un constructor care să initializeze obiectul curent de tip `Room` cu valorile altui obiect de acest tip ("constructor de copiere"). Evident, dacă acest constructor nu există, se va utiliza unul dintre constructorii existenți, eventual împreună cu metode de tip `set/get`.
- În concluzie, compoziția și agregarea sunt relații de tip `HAS_A`, care se folosesc în momentul în care dorim să reutilizăm o clasă existentă, dar nu există o relație de tipul `IS_A` între ea și noua clasă, deci nu putem să utilizăm moștenirea.
- Cu alte cuvinte, dacă noua clasă este asemănătoare, din punct de vedere al modelarii, cu o clasă definită anterior, atunci se va utiliza extinderea, realizându-se o specializare a sa prin redefinirea unor metode. Dacă noua clasă nu este asemănătoare cu o clasă deja definită, dar are nevoie de metodele sale (fără a le modifica!), atunci se va utiliza compoziția sau agregarea.
- Agregarea se va utiliza în cazul în care obiectul container nu poate controla complet obiectul asociat (extern), acesta fiind creat/modificat de alte obiecte, iar compoziția se va utiliza când obiectul container trebuie să aibă control complet asupra obiectului asociat (dacă nu furnizăm metode de acces pentru obiectul asociat, atunci nimeni din exterior nu-l poate modifica).