

EXTINDEREA CLASELOR

Cum se definesc clasele extinse

Limbajul Java pune la dispoziție și o altă facilitate importantă legată de OOP : *posibilitatea de extindere a claselor*. Pe scurt (deci incomplet) aceasta constă în:

- o clasă poate fi extinsă, adăugându-se noi câmpuri și noi metode, care permit considerarea unor atribute suplimentare și unor operații noi (asupra câmpurilor "inițiale", dar și asupra câmpurilor nou adăugate);

- unele metode ale clasei pe care o extindem pot fi redefinite, iar anumite câmpuri ale acestei clase pot fi "ascunse";

- **un obiect având ca tip clasa extinsă poate fi folosit oriunde este așteptat un obiect al clasei care a fost extinsă.**

Observație. Este total nerecomandată modificarea unei clase; actualizarea ei trebuie făcută prin mecanismul de extindere a claselor, prezentat în continuare.

Exemplu Să presupunem că dorim să urmărim mișcarea unui punct în plan. Vom începe prin a considera clasa Punct:

```
class Punct {
    int x,y;
    Punct(int x, int y) {
        this.x=x;
        this.y=y;
    }
    Punct(int x) {
        this(x, x); //apel de constructor
    }
    void origine() { x=0; y=0; }
    void simetricOx() { y=-y; }
    void simetricOy() { x=-x; }
    Punct miscare(int dx, int dy) {
        return new Punct(x+dx,y+dy);
    }
    public String toString(){
        return "("+x+", "+y+")";
    }
}
```

Vom extinde clasa Punct astfel încât punctul să aibă și o culoare:

```

class Pixel extends Punct{
    String culoare; //mosteneste x si y

    Pixel(int x, int y, String culoare){
        super(x,y);
        this.culoare=culoare;
    }

    Pixel(int x, String culoare){
        this(x,x,culoare);
    }

    /*Pixel(){} - invoca implicit constructorul fara parametrii
    din Punct() - super()*/

    void origine(){//redefinita
        super.origine();
        culoare="alb";
    }

    Pixel miscare(int dx, int dy){//diferita tip returnat
        return new Pixel(x+dx,y+dy,culoare);
    }

    public String toString(){//redefinita
        return super.toString()+" culoare "+culoare;
    }
}

class ExempluPunctPixel{
    public static void main(String arg[]){
        Punct p=new Punct(5);
        System.out.println(p);
        System.out.println(p.miscare(-1,3));
        //miscare returneaza un Punct

        Pixel px=new Pixel(2,3,"rosu");
        System.out.println(px);
        System.out.println(px.miscare(-1,3));
        //miscare returneaza acum un Pixel

        px.simetricOx();//mostenita de la Punct
        System.out.println(px);
        px.origine();
        System.out.println(px);
    }
}

```

Clasa `Pixel` moștenește câmpurile `x`, `y` și metodele `simetricOx`, `simetricOy`.

Este adoptată următoarea terminologie: clasa `Punct` este **superclasă** a lui `Pixel`, iar `Pixel` este **subclasă** (clasă extinsă) a lui `Punct`.

La invocarea constructorului unei subclase este invocat și un constructor al superclasei, explicit, cu `super(...)` **sau implicit - constructorul fără argumente**. Apelurile **`super`** din constructorii subclasei **trebuie să apară ca primă acțiune**.

În exemplul de mai sus, apelurile `super.origine()`, `super.toString()` și `super(x, y)` se referă respectiv la metoda `origine`, la metoda `toString` și la constructorul din superclasa `Punct` a lui `Pixel`.

În general putem folosi cuvântul cheie `super` sub una dintre formele:

- `super(...)` : pentru a **invoca un constructor al superclasei clasei curente**;
- `super.met(...)` : pentru a invoca o metodă a superclasei, metodă ce a fost redefinită în clasa curentă;
- `super.c` : pentru a accesa un câmp `c` al superclasei, câmp ce a fost ascuns în clasa curentă prin redeclararea sa.

Observăm că la accesarea unui câmp sau a unei metode, `super` **acționează ca referință la obiectul curent ca instanțiere a superclasei sale**.

Fie `Sub` o clasă ce extinde clasa `Super`. Sunt importante următoarele precizări:

- obiectele de tip `Sub` pot fi folosite oriunde este așteptat un obiect de tipul `Super`.
- o clasă poate fi subclasă a unei singure clase (*moștenire simplă*); nu există *moștenire multiplă* ca în alte limbaje (există însă mecanisme pentru a simula această facilitate și anume *interfețele*);
- o metodă din superclasă poate fi rescrisă în subclasă (!! folosind **aceeași semnătură**). Accesul la metoda cu aceeași semnătură din superclasă se face prin precalificare cu **`super`**;
- un câmp redeclarat în `Sub` *ascunde* câmpul cu același nume din `Super`;
- câmpurile neascunse și metodele nerescrise sunt automat *moștenite* de subclasă (în funcție de modificatorii lor de acces, despre care vom discuta ulterior)

Observație Și metoda `miscare` este rescrisă în clasa `Pixel`, deși nu are același tip returnat cu cea din clasa `Punct`. Dacă până în versiunea 5 o metodă declarată într-o clasă poate fi rescrisă într-o subclasă prin declararea ei în subclasă cu același nume, aceeași semnătură și același tip al valorii întoarse, din versiunea 5 doar numele și semnatura trebuie să fie aceleași, **tipul returnat fiind necesar doar să fie un subtip al celui returnat de metoda inițială**. Astfel, am putut rescrie în clasa `Pixel` metoda (din clasa `Punct`)

miscare cu signatura (int, int) și care returna un obiect de tip Punct astfel încât ea să returneze un obiect de tip Pixel (Pixel este subclasă a clasei Punct).

Inițializare și constructori

Prima acțiune întreprinsă la crearea unui obiect constă în a atribui câmpurilor valorile inițiale implicite:

- 0 - pentru tipuri numerice
- `\u0000` - pentru tipul **char**
- `false` - pentru tipul **boolean**
- **null** - pentru referințe (la obiecte, inclusiv tablouri și `String`).

În continuare este invocat constructorul corespunzător (determinat de semnătură, în același mod ca pentru metode); acțiunea sa cuprinde trei faze:

- 1) Invocarea unui constructor al superclasei și anume:
 - explicit, cu **super** (. . .) ;
 - implicit: constructorul fără argumente.
- 2) Executarea, în ordinea în care apar, a inițializărilor directe ale câmpurilor;
- 3) Executarea corpului constructorului.

Exemplu Să considerăm clasele:

```
class A {
    int va=1, v;
    A() {
        v=va;
    }
}

class B extends A {
    int vb=2;
    B() {
        v=va+vb;
    }
}
```

La crearea unui obiect de tipul B, prin **new** B() , au loc în ordine următoarele acțiuni:

- câmpurile `v`, `va`, `vb` primesc valoarea implicită 0;
- este invocat constructorul B;
- este invocat constructorul A (al superclasei);
- are loc inițializarea directă a lui `va`, care primește valoarea 1;

- este executat corpul constructorului A și ca urmare v primește valoarea 1;
- are loc inițializarea directă a lui $v.b$, care primește valoarea 2;
- este executat corpul constructorului B și ca urmare v primește valoarea 3.

Rescrierea metodelor și ascunderea câmpurilor. Polimorfism și legare dinamică

Câmpurile pot fi *ascunse* prin redeclararea lor într-o subclasă. Câmpul cu același nume din superclasă nu mai poate fi accesat direct prin numele său, dar ne putem referi la el folosind **super** sau o referință la tipul superclasei.

O metodă declarată într-o clasă poate fi rescrisă într-o subclasă prin declararea ei în subclasă cu același nume, aceeași semnătură și același tip al valorii întoarse (sau tipul returnat este un subtip al celui returnat de metoda inițială, după cum am amintit anterior).

Spunem că metodele rescrise sunt *ascunse* (dacă e vorba de metode statice) sau *redefinite* (în cazul metodelor nestatice). Nu este vorba numai de o diferență de terminologie (după cum vom vedea în cele ce urmează); metodele statice pot fi rescrise (ascunse) numai de metode statice, iar metodele nestatice pot fi rescrise (redefinite) numai de metode nestatice.

Fie A o clasă și fie B o subclasă a sa. Să considerăm următoarele patru acțiuni echivalente din punctul de vedere al obiectului a ce ia naștere:

- 1) A a; a = **new** B (...);
- 2) A a = **new** B (...);
- 3) A a; B b; b = **new** B (...); a = b;
- 4) A a; B b; b = **new** B (...); a = (A) b;

Să observăm că este vorba de o conversie de la o subclasă la o superclasă, conversie numită **upcasting**; ea poate fi implicită (ca în primele trei acțiuni) sau explicită (ca în cea de a patra acțiune).

Vom spune că obiectul a are **tipul declarat** A și **tipul real** B, ceea ce pune în evidență noțiunea de **polimorfism**. Tipul real al unui obiect coincide cu tipul său declarat (este cazul obiectului b) sau este o subclasă a tipului declarat (vezi obiectul a).

Fie `camp` un câmp al clasei A, ce este redeclarat (ascuns) în subclasa B. Dacă obiectul a face referire la câmpul `camp`, atunci este vorba de câmpul declarat în clasa A, adică **este folosit tipul declarat al obiectului**.

Fie `met` o metodă a clasei A, care este rescrisă în subclasa B. **La invocarea metodei `met` de către obiectul a, este folosită fie implementarea corespunzătoare**

metodei ascunse (dacă este statică), fie cea corespunzătoare metodei redefinite (dacă este nestatică). Cu alte cuvinte, pentru metode statice este folosit tipul declarat (la fel ca pentru câmpuri), iar pentru metode nestatice este folosit tipul real al obiectului.

Constructorii unei clase nu sunt considerați membri ai clasei. De aceea ei nu pot fi moșteniți sau redeclarați.

Exemple

1. Justificați rezultatele afișate de următorul program:

```
class Super{
    int camp;

    Super(int camp){
        this.camp=camp;
        metNestaticaDinConstructor();
    }

    static void metStatika() {
        System.out.println("static_Super");
    }

    void metNestatica() {
        System.out.println("Super");
    }

    void metNestaticaDinConstructor(){
        System.out.println("Din Constructor Super");
    }
}

class Sub extends Super {
    char camp;
    Sub(int camp){
        super(camp);
        this.camp=(char) camp;
        metNestaticaDinConstructor();
    }

    static void metStatika() {
        System.out.println("static_Sub");
    }

    void metNestatica(){
        System.out.println("Sub");
    }

    void metNestaticaDinConstructor(){
        System.out.println("Din Constructor Sub");
    }
}
```

```

class Test {
    public static void main(String[] s){
        Super ob=new Sub(97);
        ob.metNestatica();
        ob.metStatica();
        System.out.println(ob.camp);
        Sub obSub=(Sub)ob;
        obSub.metNestatica();
        obSub.metStatica();
        System.out.println(obSub.camp);
    }
}

```

produce la ieşire:

```

Din Constructor Sub
Din Constructor Sub
Sub
static_Super
97
Sub
static_Sub
a

```

2. Adăugați în ExempluPunctPixel următorul cod și justificați rezultatele afișate

```

p=px;//cast implicit la Punct
System.out.println(p);
System.out.println(p.miscare(2,3));
//metoda miscare tot a clasei Pixel, va intoarce un Pixel

```

Să considerăm o clasă C și două subclase X și Y ale sale, precum și următoarea secvență de instrucțiuni:

```

C ob;
. . . ob = new X(...);
. . . ob = new C(...);
. . . ob = new Y(...);
. . .

```

în care ob are mai întâi tipul real X, apoi tipul real C, apoi tipul real Y. Spunem că ob este o variabilă **polimorfică**, deoarece are pe rând forma (comportamentul) a mai multor clase.

Polimorfismul constă în posibilitatea ca o invocare ob.met(...) să aibă drept consecință invocarea unei anumite metode cu numele met, în funcție de tipul real al obiectului ob.

Legarea dinamică identifică metoda `met` respectivă la executarea programului. Polimorfismul și legarea dinamică sunt două caracteristici esențiale ale programării orientate pe obiecte.

Java folosește **legarea dinamică**, adică identificarea metodei nestatice concrete care este folosită la o invocare se face la executare și nu la compilare. De aceea legarea dinamică se mai numește **legare târzie**.

Necesitatea legării dinamice apare din următorul exemplu:

Exemplu Fie A o clasă în care este definită o metodă `met`, iar B o subclasă a sa în care metoda `met` este redefinită. Secvența de instrucțiuni:

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

arată că doar la momentul invocării metodei devine clar care dintre cele două metode `met` va fi invocată.

Despre modificatori

Lista principalilor modificatorilor folosiți în Java este următoarea:

- modificatorii de acces (**public**, **protected**, **private** și cel implicit);
- **abstract**, **static**, **final**, **synchronized**, **native**, **transient**, **volatile**.
- din versiunea 8: **default** (pentru metode din interfețe)

Ei pot fi folosiți astfel:

- pentru *clase*: **public**, cel implicit, **abstract**, **final**;
- pentru *interfețe*: modificatorii de acces;
- pentru *constructori*: modificatorii de acces;
- pentru *câmpuri*: modificatorii de acces, **final**, **static**, **transient**, **volatile**;
- pentru *metode*: modificatorii de acces, **final**, **static**, **abstract**, **synchronized**, **native**.

▪ Modificatorii de acces

Modificatorii de mai sus joacă un rol nu numai în privința accesului, dar și pentru facilitarea de extindere a claselor, ceea ce va face diferența între modificatorul `protected` și cel implicit.

Modificator	<i>Membrii clasei cu acest modificator sunt accesibili ...</i>	<i>Membrii clasei cu acest modificator ...</i>
public	de oriunde clasa este accesibilă	sunt moșteniți în subclase
protected	din cod din același pachet și din subclasele clasei respective	sunt moșteniți în subclase
Implicit (<i>package</i>)	din cod din același pachet	sunt moșteniți numai în subclase din același pachet
private	numai din interiorul clasei	nu sunt moșteniți în subclase

Dacă o clasă este declarată cu modificatorul `public`, constructorul implicit are automat acest modificator.

În metoda rescrisă **putem schimba modificatorul de acces**, cu condiția ca dreptul de acces să crească.

▪ Modificatorul `static`

Modificatorul **`static`** poate fi folosit la declararea câmpurilor și metodelor unei clase, precum și pentru a anunța un bloc de inițializare static.

O variabilă locală **nu** poate fi declarată cu **`static`**.

Fie `c` un câmp declarat cu **`static`** în clasa `Clasa`. Această declarație are două consecințe:

- câmpul `c` este comun tuturor obiectelor de tipul `Clasa`;
- referirea la câmp din exteriorul clasei se face fie conform regulii generale (prin crearea și utilizarea unei instanțe a clasei `Clasa`), fie direct, prin precalificarea câmpului cu numele clasei (`Clasa.c`).

Inițializarea câmpurilor statice se face la inițializarea clasei.

Fie acum `met` o metodă declarată cu modificatorul **`static`** în clasa `Clasa`; ea se numește *metodă statică* sau *metodă de clasă*. Invocarea ei se poate face fie conform regulii generale (prin crearea și utilizarea unei instanțe a clasei `Clasa`), fie direct, prin precalificarea metodei cu numele clasei: `Clasa.met(...)` ;

De aceea într-o metodă statică nu poate fi folosită referința **`this`**.

O metodă statică poate accesa numai câmpurile și metodele statice ale clasei din care face parte sau a altei clase.

▪ Modificatorul **final**

Modificatorul **final** poate fi atașat câmpurilor, dar și variabilelor locale.

În cazul variabilelor locale, variabilei i se atribuie valoare la declarare sau înainte de a fi folosită; variabila nu poate primi de două ori valoare. Menționăm că **final** este singurul modificator ce poate fi atașat unei variabile locale.

Unui câmp **final** trebuie să i se atribuie valoare fie prin inițializare, fie prin constructori.

Menționăm că în general modificatorul **final** este folosit împreună cu **static**, pentru a trata un câmp ca o "constantă cu nume".

Modificatorul **final** poate fi asociat (în afara variabilelor locale și câmpurilor) și metodelor și claselor. El trebuie înțeles în sensul de "variantă finală":

- o metodă cu acest modificator **nu poate fi rescrisă** într-o subclasă;
- o clasă cu acest modificator **nu poate fi extinsă**.

CLASE ABSTRACTE

Metode și clase abstracte

Se întâmplă frecvent ca atunci când lucrăm cu clase extinse, pentru o clasă să nu putem preciza implementarea unei metode, deoarece în subclasele sale ea va fi specifică fiecăreia dintre ele. Atunci metodele din această categorie vor fi marcate cu cuvântul cheie **abstract** și se vor reduce la antetul lor; în același timp și clasa trebuie însoțită de atributul **abstract**. Fiecare metodă abstractă trebuie implementată, adică (re)definită în orice subclasă care nu este la rândul său abstractă.

Nu este posibil să creăm obiecte ca instanțe ale unei clase abstracte. Pe de altă parte, într-o subclasă putem să redefinim o metodă a superclasei transformând-o într-o metodă abstractă; aceasta are sens de exemplu dacă în subarborele având ca rădăcină subclasa, comportamentul materializat prin acea clasă nu mai este valabil în subarbore și este specific fiecărei extensii a subclasei.

Exemplu Următoarea clasă are ca scop să măsoare timpul necesitat de executarea unei metode oarecare fără parametri și care nu întoarce vreun rezultat:

```

abstract class C {
    abstract void met();
    public long timp() {
        long t0 = System.currentTimeMillis();
        met();
        return System.currentTimeMillis()-t0;
    }
}

```

unde a fost folosită metoda statică `currentTimeMillis()` a clasei `System`, metodă care întoarce timpul curent în milisecunde. Într-o clasă ce extinde `C`, se poate redefini metoda `met` și apoi, folosind un obiect ce este o instanțiere a noii clase, putem apela metoda `timp` pentru a determina timpul de executare a metodei `met`, ce are acum o implementare precisă:

```

class C1 extends C {
    void met () {
        long x=0;
        for (int i=0; i<1000000; i++) x+=i;
    }
}

class C2 extends C {
    void met () {
        Long x=0L;
        for (int i=0; i<1000000; i++) x+=i;
    }
}

class Abstr {
    public static void main(String[] s) {
        C1 ob1 = new C1();
        C ob2;//tip clasa abstracta
        ob2 = new C2();
        System.out.println( "durata="+ob1.timp() );
        System.out.println( "durata="+ob2.timp() );
    }
}

```

Legat de exemplul de mai sus, este posibil ca la executări diferite să obținem timpi diferiți: aceasta este o consecință a faptului că pe perioada executării programului nostru sistemul gazdă "mai face și altceva", sau că a intervenit colectorul de reziduuri.

CLASE INTERNE

Java permite să definim clase ca membri ai unei clase. Avantajul utilizării claselor interne constă în primul rând în faptul că permite includerea lor exact acolo unde trebuie să apară din punct de vedere logic, dar și în ușurința folosirii lor.

Din interiorul unei clase interne **putem să ne referim direct la câmpurile clasei ce o conține**. O clasă internă nu poate fi folosită direct decât în clasa în care a fost declarată și în descendenții acesteia. Domeniul ei de vizibilitate este cel al unui câmp.

Exemplu

```
class Lista{
    int nr;
    class Nod{
        int info;
        Nod urm;
        Nod(){nr++; }
        Nod(int x){info=x; nr++;}
    }
    Nod cap;
    void add(int x){
        Nod p=new Nod(x);
        if (cap == null) cap=p;
        else p.urm=cap;
        cap=p;
    }
    void afis(){
        Nod p=cap;
        while(p!=null){System.out.println(p.info); p=p.urm;}
    }
    public static void main(String[] w) {
        Lista l=new Lista();
        l.add(3); l.add(7); l.afis();
        System.out.println(l.nr);
    }
}
```

Menționăm că există și clase fără nume (anonime). O *clasă anonimă* apare într-o expresie de forma:

new SuperA(...) {...}

prin care clasa anonimă extinde o clasă sau implementează o interfață SuperA și prin care este creat un obiect al clasei anonime.

Observație. Crearea de clase anonime permite *extinderea implicită a unei clase și implementarea implicită a unei interfețe* (fără a folosi **extends** sau **implements**). Drept urmare codul devine mai scurt și mecanismul este folosit "la locul potrivit". Această tehnică este larg folosită de exemplu în lucrul cu interfețe grafice.

Exemplu - Extinderea implicită a unei clase

```
abstract class A {
    A(int i) { System.out.println(i); }
    void met() { };
}

class AExt {
    static A obA;
    public static void main(String[] s) {
        obA = new A(3) {
            void met() { System.out.println("O.K."); }
        };
        obA.met();
    }
}
```

Exemplu - Implementarea implicită a unei interfețe

```
public static void main(String[] w){
    ArrayList<Integer> a=new ArrayList<Integer>();
    a.add(23); a.add(7); a.add(3);
    Collections.sort(a, new Comparator<Integer>() {
        public int compare(Integer x, Integer y){
            if (x<y) return -1;
            if (x>y) return 1;
            return 0; }    });
    System.out.println(a);
}
```