EXTINDEREA CLASELOR

Cum se definesc clasele extinse

Limbajul Java pune la dispoziţie şi 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 şi noi metode, care permit considerarea unor atribute suplimentare şi unor operaţii noi (asupra câmpurilor "iniţiale", dar şi 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 aşteptat un obiect al clasei care a fost extinsă.**

Observaţie. **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 mişcarea 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 şi 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){//difera 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` moşteneşte câmpurile `x,y` şi 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 şi un constructor al superclasei, explicit, cu **`super`**`(...)` **sau implicit - constructorul fără argumente.** Apelurile **`super`** din constructorii subclasei **trebuie să apară ca primă acţiune.**

În exemplul de mai sus, apelurile `super.origine()`, `super.toString()` şi `super(x,y)` se referă respectiv la metoda `origine`, la metoda `toString` şi 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` **acţionează ca referinţă la obiectul curent ca instanţiere a superclasei sale.**

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

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

Observaţie Şi metoda `miscare` este rescrisă în clasa `Pixel`, deşi nu are acelaşi 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 acelaşi nume, aceeaşi signatură şi acelaşi tip al valorii întoarse, din versiunea 5 doar numele şi signatura trebuie sa fie aceleaşi, **tipul returnat fiind necesar doar sa fie un subtip al celui returnat de metoda iniţială.** Astfel, am putut rescrie în clasa `Pixel` metoda (din clasa `Punct`)

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

Iniţializare şi constructori

Prima acţiune întreprinsă la crearea unui obiect constă în a atribui câmpurilor valorile iniţiale implicite:

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

În continuare este invocat constructorul corespunzător (determinat de signatură, în acelaşi mod ca pentru metode); acţiunea sa cuprinde trei faze:

1) Invocarea unui constructor al superclasei şi anume:
   - explicit, cu **`super`**`(...)` ;
   - implicit: constructorul fără argumente.
2) Executarea, în ordinea în care apar, a iniţializărilor 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 acţiuni:

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

- este executat corpul constructorului `A` şi ca urmare `v` primeşte valoarea `1`;
- are loc iniţializarea directă a lui `vb`, care primeşte valoarea `2`;
- este executat corpul constructorului `B` şi ca urmare `v` primeşte valoarea `3`.

Rescrierea metodelor şi ascunderea câmpurilor. Polimorfism şi legare dinamică

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

O metodă declarată într-o clasă poate fi rescrisă într-o subclasă prin declararea ei în subclasă cu acelaşi nume, aceeaşi signatură şi acelaşi tip al valorii întoarse (sau tipul returnat este un subtip al celui returnat de metoda iniţială, 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 diferenţă 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 şi fie `B` o subclasă a sa. Să considerăm următoarele patru acţiuni echivalente din punctul de vedere al obiectului `a` ce ia naştere:

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 acţiuni) sau explicită (ca în cea de a patra acţiune).

Vom spune că obiectul `a` are **tipul declarat** `A` şi **tipul real** `B`, ceea ce pune în evidenţă noţiunea 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 consideraţi membri ai clasei. De aceea ei nu pot fi moşteniţi sau redeclaraţi.

Exemple

1. Justificaţi rezultatele afişate de următorul program:

```
class Super{
    int camp;

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

    static void metStatica() {
        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 metStatica() {
        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. Adăugaţi în `ExempluPunctPixel` următorul cod şi justificaţi rezultatele afişate

```
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` şi două subclase `X` şi `Y` ale sale, precum şi următoarea secvenţă de instrucţiuni:

```
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 consecinţă invocarea unei anumite metode cu numele `met`, în funcţie de tipul real al obiectului `ob`.

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

Java foloseşte ***legarea dinamică***, adică identificarea metodei nestatice concrete care este folosită la o invocare se face la executare şi nu la compilare. De aceea legarea dinamică se mai numeşte ***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. Secvenţa de instrucţiuni:

```
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 modificatorilor folosiţi în Java este următoarea:
- modificatorii de acces (**`public`**, **`protected`**, **`private`** şi cel implicit);
- **`abstract`**, **`static`**, **`final`**, **`synchronized`**, **`native`**, **`transient`**, **`volatile`**.

Ei pot fi folosiţi astfel:

- pentru *clase*: **`public`**, cel implicit, **`abstract`**, **`final`**;
- pentru *interfeţe*: 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 privinţa accesului, dar şi pentru facilitatea de extindere a claselor, ceea ce va face diferenţa între modificatorul `protected` şi cel implicit.

Modificator	*Membrii clasei cu acest modificator sunt accesibili ...*	*Membrii clasei cu acest modificator ...*
`public`	de oriunde clasa este accesibilă	sunt moşteniţi în subclase
`protected`	din cod din acelaşi pachet şi din subclasele clasei respective	sunt moşteniţi în subclase
Implicit (*package*)	din cod din acelaşi pachet	sunt moşteniţi numai în subclase din acelaşi pachet
`private`	numai din interiorul clasei	nu sunt moşteniţi 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 condiţia ca dreptul de acces să crească.

- **Modificatorul `static`**

Modificatorul **`static`** poate fi folosit la declararea câmpurilor şi metodelor unei clase, precum şi pentru a anunţa un bloc de iniţializare static.

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

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

- câmpul `c` este comun tuturor obiectelor de tipul `Clasa`;

- referirea la câmp din exteriorul clasei se face fie conform regulii generale (prin crearea şi utilizarea unei instanţe a clasei `Clasa`), fie direct, prin precalificarea câmpului cu numele clasei (`Clasa.c`).

Iniţializarea câmpurilor statice se face la iniţializarea clasei.

Fie acum `met` o metodă declarată cu modificatorul **`static`** în clasa `Clasa`; ea se numeşte *metodă statică* sau *metodă de clasă*. Invocarea ei se poate face fie conform regulii generale (prin crearea şi utilizarea unei instanţe a clasei `Clasa`), fie direct, prin precalificarea metodei cu numele clasei: `Clasa.met(...)` ;
De aceea într-o metodă statică nu poate fi folosită referinţa **`this`**.

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

- **Modificatorul `final`**

Modificatorul **`final`** poate fi ataşat câmpurilor, dar şi 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. Menţionăm că **`final`** este singurul modificator ce poate fi ataşat unei variabile locale.

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

Menţionăm 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 şi câmpurilor) şi metodelor şi claselor. El trebuie înţeles 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 şi 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`** şi se vor reduce la antetul lor; în acelaşi timp şi clasa trebuie însoţită 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 instanţe 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 şi este specific fiecărei extensii a subclasei.

Exemplu Următoarea clasă are ca scop să măsoare timpul necesitat de executarea unei metode oarecare fără parametri şi 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` şi apoi, folosind un obiect ce este o instanţiere 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 obţinem timpi diferiţi: aceasta este o consecinţă a faptului că pe perioada executării programului nostru sistemul gazdă "mai face şi 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 şi în uşurinţa folosirii lor.

Din interiorul unei clase interne putem să ne referim direct la câmpurile clasei ce o conţine. O clasă internă nu poate fi folosită direct decât în clasa în care a fost declarată şi în descendenţii 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);
      }
}
```

Menţionăm că există şi 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 interfaţă `SuperA` şi prin care este creat un obiect al clasei anonime.

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