Lucrarea 4

Moştenire şi polimorfism

Cuprins

Tehnologia orientată pe obiecte permite extinderea comportamentului unei clase existente prin definirea unei clase noi, care moşteneşte conţinutul primei clase, adăugând la acesta elementele specifice ei. Clasa moştenită se numeşte clasă de bază sau superclasă, iar clasa care realizează extinderea se numeşte subclasă, clasă derivată, sau clasă descendentă. Relaţia de moştenire este o relaţie de forma "*este un fel de"*, adică "*subclasa este un fel de superclasă"*. Practic, relaţia de moştenire între clase este o reflectare a ierarhizării existente între entităţile modelate de clasele respective.

Relaţia de moştenire prezintă două aspecte esenţiale:

- reutilizare de cod
- polimorfism.

Relaţia de moştenire în Java

În Java, relaţia de moştenire dintre o superclasă şi o subclasă se exprimă astfel:

```
class nume_subclasa extends nume_superclasa
{
        //conţinut specific subclasei
}
```

Se consideră clasa Poligon şi o clasă Dreptunghi care moşteneşte clasa Poligon:

```
class Poligon {
        protected double[ ] laturi;
        public Poligon(int n) { laturi = new double[n] }
        public double perimetru( ) {
            double s=0;
            for(int i=0;i<laturi.length;i++) s+=laturi[i];
            return s;
        }
}
class Dreptunghi extends Poligon {
        public Dreptunghi(double L, double h){
            super(4);
            laturi[0] = laturi[2] = L;
```

```
            laturi[1] = laturi[3] = h;
        }
        public double aria( ){
            return laturi[0]*laturi[1];
        }
}
```

Clasa Dreptunghi reutilizează codul definit în clasa Poligon, la care adaugă elemente proprii. Pe lângă Dreptunghi, am putea defini şi alte clase, cum ar fi Triunghi, Patrat, Pentagon etc. care să moştenească Poligon. Superclasa este *factorul comun* al subclaselor sale sau, altfel spus, *codul factorului comun este reutilizat în subclase*.

Despre relaţia de moştenire în Java se pot spune următoarele:

- o subclasă poate extinde o singură superclasă (moştenire simplă)
- o superclasă poate fi moştenită de mai multe subclase distincte
- o subclasă poate fi superclasă pentru alte clase
- o clasă de bază împreună cu toate descendentele ei formează o ierarhie de clase.

Reguli de vizibilitate în contextul relaţiei de moştenire

Regulile de vizibilitate vor fi discutate din două perspective:

- accesul funcţiilor unei subclase la membrii moşteniţi de la superclasele ei
- accesul clienţilor unei subclase la membrii ei, moşteniţi sau specifici

Accesul funcţiilor unei subclase la membrii moşteniţi:

Pentru a ilustra drepturile de acces vom lua următorul exemplu:

```
class SuperClasa {
        private int priv;
        protected int prot;
        public int pub;
}
class SubClasa extends SuperClasa {
        private int priv_local;
        protected int prot_local;
        public int pub_local;
        public void metoda(SubClasa p ) {
            priv = 1; //eroare
            prot = 2; //corect
            pub = 3; //corect
            priv_local = 4; //corect
            prot_local = 5; //corect
            pub_local = 6; //corect
            p.priv = 1; //eroare
            p.prot = 2; //corect
            p.pub = 3; //corect
            p.priv_local = 4; //corect
```

```
            p.prot_local = 5; //corect
            p.pub_local = 6; //corect
        }
}
```

În metodele specifice ale unei subclase pot fi referiţi acei membri moşteniţi, care în superclasă au fost precedaţi de modificatorii *protected* sau *public*. Membrii moşteniţi, care în superclasă sunt precedaţi de modificatorul *private*, deşi fizic fac parte din subclasă, nu sunt accesibili în metodele acesteia. Referirea membrilor moşteniţi se face direct, folosind numele lor, la fel ca referirea membrilor specifici.

Acest lucru este foarte normal, deoarece şi membrii moşteniţi sunt conţinuţi în subclasă, numai că ei nu au fost declaraţi explicit. Includerea lor se face automat, ca efect al clauzei extends. Acest lucru nu este valabil şi pentru constructori. Revenind la exemplul cu clasele *Poligon — Dreptunghi*, trebuie spus că un obiect al clasei *Dreptunghi* conţine doi constructori: unul moştenit, responsabil cu iniţializarea porţiunii moştenite, şi unul local, responsabil cu iniţializarea porţiunii specifice subclasei. Referirea în subclasă la constructorul moştenit se face folosind cuvântul cheie *super*, aşa cum se observă în constructorul clasei *Dreptunghi*.

Cuvântul *super* este un omolog al lui *this* şi reprezintă referinţa la partea moştenită a unui obiect. În afară de apelul constructorului moştenit, cuvântul *super* se va utiliza pentru a distinge între un membru local al subclasei şi un membru moştenit, ambii cu acelaşi nume.

Accesul clienţilor unei subclase la membrii acesteia

Exemplului din paragraful anterior i se adaugă secvenţa următoare:

```
class Client {
        public void oFunctie( ) {
            SuperClasa sp = new SuperClasa();
            SubClasa sb = new SubClasa();
            sp.priv = 1; //eroare
            sp.prot = 2; //corect, doar dacă Client şi  SuperClasa s-ar afla in acelaşi pachet
            sp.pub = 3; //corect
            sb.priv = 1; //eroare
            sb.prot = 2; //corect, doar dacă Client şi SubClasa s-ar afla în acelaşi pachet
            sb.pub = 3; //corect
            sb.priv_local = 4; //eroare
            sb.prot_local = 5; //corect, doar dacă Client şi SubClasa s-ar afla în acelaşi pachet
            sb.pub_local = 6; //corect
        }
}
```

În clienţii unei subclase pot fi referiţi acei membri, moşteniţi sau locali, care sunt precedaţi de modificatorul public. Membrii declaraţi ca *protected* sunt accesibili numai în situaţia în care clientul se află în acelaşi pachet cu subclasa respectivă. La referirea membrilor unei subclase de către client, nu se face deosebire între membrii moşteniţi şi cei locali.

Constructorii şi moştenirea

În exemplul anterior se observă că cele două clase nu au constructori expliciţi, constructorul SubClasa() fiind de forma:

```
public SubClasa() { super( ); }
```

Constructorul subclasei apelează constructorul *no-arg* al superclasei. Dacă pentru *SuperClasa* am fi definit un constructor *no-arg* explicit, atunci acesta ar fi fost cel apelat de constructorul *SubClasa*. Dacă ar fi existat un constructor oarecare în *SubClasa*, iar acesta nu ar fi apelat explicit vreun constructor din *SuperClasa*, în mod automat se încearcă apelul constructorului *no-arg super()*, înainte de a se executa instrucţiunile din constructorul *SubClasa*:

```
public SubClasa(. . .) {
        super(); //apel implicit
        //alte instrucţiuni prevăzute de programator
}
```

Dacă *SuperClasa* nu are constructor *no-arg*, apelul implicit de mai sus ar genera eroare. Prin urmare, dacă programatorul defineşte constructori expliciţi într-o subclasă, el trebuie să aibă grijă ca aceşti constructori să apeleze explicit constructorii adecvaţi ai superclasei:

```
super (parametri-actuali)
```

Un asemenea apel trebuie să fie prima instrucţiune din constructorul subclasei. Constructorul unei subclase se defineşte astfel încât lista lui de parametri să se compună dintr-un set de parametri necesari iniţializării câmpurilor moştenite (transmis constructorului super) şi un alt set, necesar iniţializării câmpurilor locale.

Execuţia constructorului se desfăşoară în 3 etape:

- apelul constructorului superclasei
- iniţializarea câmpurilor cu valori date la declarare şi execuţia blocurilor de iniţializare, unde este cazul
- execuţia corpului propriu-zis al constructorului.

Operatorul *instanceof*

Prin intermediul unei referinţe de superclasă putem indica şi utiliza şi obiecte ale subclaselor ei. Operatorul instanceof este util atunci când trebuie să cunoaştem clasa concretă de care aparţine un obiect referit prin intermediul unei referinţe a unui supertip. Aplicarea operatorului se face printr-o expresie de forma:

```
referinta_obiect instanceof nume_clasa
```

Redefinirea metodelor

Relaţia de moştenire poate fi utilizată atunci când o anumită clasă (subclasă) extinde comportamentul altei clase (superclase), în sensul că superclasa înglobează aspectele comune ale unei ierarhii de abstracţiuni, iar subclasele adaugă la această parte comună funcţiuni specifice. De asemenea, o referinţă la superclasă poate indica şi manipula obiecte ale oricărei clase descendente din ea. Manipularea se referă la faptul că, prin intermediul acelei referinţe se pot apela metodele definite în superclasă, indiferent dacă obiectul concret aparţine superclasei sau uneia dintre subclase. Această facilitate constituie aşa-numitul polimorfism parţial. Folosind o referinţă a superclasei nu se poate, însă, apela o metodă locală a subclasei, decât dacă se realizează o conversie explicită *(casting)* a referinţei, la tipul subclasei. În Java relaţia de moştenire poate fi aplicată şi în cazul în care se doreşte ca, pe lângă extinderea comportamentului, să realizăm şi o adaptare (specializare) a lui, în sensul că unele metode care în superclasă au o anumită implementare, în subclase să aibă o altă implementare, adaptată la cerinţele locale ale subclasei.

```
package exemplu1;

class Angajat {
	protected String nume;
	protected double sal_ora;

	public Angajat(String nume, double sal_ora) {
		this.nume = nume;
		this.sal_ora = sal_ora;
	}

	public double calcSalar(int nr_ore) {
		return (sal_ora * nr_ore);
	}

	public String toString() {
		return nume;
	}
}
class Sef extends Angajat {
	private double proc_cond; // procent de indemnizatie conducere

	public Sef(String nume, double sal_ora, double proc_cond) {
		super(nume, sal_ora);
		this.proc_cond = proc_cond;
	}

	public double calcSalar(int nr_ore) {
		return ((1 + proc_cond / 100) * sal_ora * nr_ore);
	}
}
```

```
class MainApp {
	public static void main(String[] args) {
		Angajat a1 = new Angajat("Artaxerse Coviltir", 100);
		Angajat a2 = new Sef("Al Bundy", 200, 33.2);
		final int ore_luna = 160;

		System.out.println("Salar " + a1 + "=" + a1.calcSalar(ore_luna));
		System.out.println("Salar " + a2 + "=" + a2.calcSalar(ore_luna));
	}
	// . . .
}
```

Se observă că aceeaşi metodă, *calcSalar()* apare atât în clasa *Angajat*, cât şi în clasa *Sef*, dar cu implementări diferite. *Atunci când într-o subclasă apare o metodă având semnătura identică cu o metoda din superclasă, se spune că metoda din subclasă o redefineşte pe omonima ei din superclasa*. În acest caz, un obiect al subclasei practic are două exemplare ale metodei respective, unul moştenit şi unul propriu. Ce se întâmplă la nivelul clienţilor ierarhiei? Se observă că în metoda *main()* din clasa *MainApp* se folosesc două referinţe la *Angajat*: una indică un obiect al clasei *Angajat*, iar cealaltă un obiect al clasei *Sef*. Prin intermediul celor două referinţe se apelează metoda *calcSalar()*. Se pune problema care dintre cele două forme ale metodei se apelează efectiv? În *Java*, regula care se aplică în asemenea cazuri este următoarea: **Se va executa întotdeauna acel exemplar de metodă care este definit în clasa la care aparţine obiectul concret indicat de referinţa utilizată.**

Cu alte cuvinte, clasa obiectului „dictează" şi nu tipul referinţei. Astfel, deşi ambele referinţe a1 şi a2 din exemplul considerat sunt de tip Angajat, ele indică, de fapt, obiecte ale unor clase diferite. Deci, apelul *a1.calcSalar(...)* va declanşa execuţia metodei *calcSalar()* definită în clasa *Angajat*, iar apelul *a2.calcSalar(...)* va declanşa execuţia metodei *calcSalar()* definită în clasa *Sef*. **Polimorfismul permite interschimbarea obiectelor indicate de o referinţă a superclasei, într-o manieră transparentă pentru clienţi.**

În *Java*, clasa concretă la care aparţine un obiect indicat de o referinţă se cunoaşte, însă, abia la execuţia programului. Deci, un apel de metodă nu este „rezolvat" (adică pus în corespondenţă cu o implementare anume) la compilare, ci doar la execuţie. Acest procedeu se numeşte ***legare dinamică*** sau amânată *(late binding)*. S-a spus mai sus că un obiect al clasei *Sef* „posedă" două metode *calcSalar()*: cea moştenită de la *Angajat* şi cea locală. Din cauză că în *Java* se aplică legarea dinamică, înseamnă că, din perspectiva clienţilor clasei *Sef* varianta locală o „eclipsează" pe cea moştenită, neexistând posibilitatea ca un client să forţeze apelul metodei moştenite (decât aplicând o soluţie „ocolitoare", adică folosind o metodă cu alt nume care să apeleze metoda *calcSalar()* moştenită).

În interiorul subclasei se poate realiza distincţia între cele două variante ale unei metode redefinite, şi anume folosind simbolul ***super***. Astfel, metoda *calcSalar()* din clasa *Sef* se poate folosi de omonima ei din clasa *Angajat*:

```
class Sef extends Angajat {
        //. . .
        public double calcSalar(int nr_ore) {
             return ((1 + proc_cond / 100) *    super.calcSalar(nr_ore));
        }
}
```

Practic, aceasta este singura situaţie în care nu se aplică legarea dinamică, ci apelul de forma *super.numeMetoda(...)* este pus în corespondenţă exact cu implementarea din superclasa clasei curente.

Legarea dinamică şi constructorii

Deoarece în Java la apelul metodelor non-statice se aplică legarea dinamică, pe de o parte, şi ţinând cont de modul în care se apelează constructorii într-o ierarhie de clase, pe de altă parte, trebuie să avem grijă cum proiectăm constructorii dacă aceştia apelează la rândul lor metode ale claselor respective.

```
package exemplu2;

class SuperClasa {
        protected int a;
        protected int rez;
        private int x;

        public SuperClasa() {
                a = 1;
                rez = 2;
                x = calcul();  //x=0
        }

        public int calcul() {
                return (rez + a);
        }

        @Override
        public String toString() {
                return "SuperClasa [a=" + a + ", rez=" + rez + ", x=" + x + "]";
        }

}

class SubClasa extends SuperClasa {
        protected int b;
        private int y;
```

```
	public SubClasa() {
		b = 3;
		y = calcul();
	}

	public int calcul() {//asta 2*0 si apoi 2 *3
		return (rez * b);
	}

	@Override
	public String toString() {
		return super.toString()+" SubClasa [b=" + b + ", y=" + y + "]";
	}
}

public class MainApp {
	public static void main(String[] args) {
		SubClasa ob = new SubClasa();
		System.out.println(ob);
	}
}
//Output: SuperClasa [a=1, rez=2, x=0] SubClasa [b=3, y=6]
```

Să urmărim ce se întâmplă la crearea unui obiect al clasei Subclasa: ţinând cont de ordinea în care au loc iniţializările câmpurilor unui obiect, înseamnă că se execută următorii paşi:

- câmpurile a, rez, x, b şi y se iniţializează cu valorile implicite corespunzătoare tipurilor lor, deci, în cazul nostru cu 0;
- se lansează constructorul *SubClasa()*; ca primă acţiune a acestuia are loc apelul constructorului no-arg *SuperClasa()*;
- în constructorul *SuperClasa()* se execută iniţializările lui *a* şi *rez* cu 1, respectiv 2, după care se apelează metoda *calcul()*. În acest moment, apelul se leagă de varianta metodei *calcul()* definită în clasa *SubClasa*, de care aparţine obiectul în curs de iniţializare. Efectul este că *x* se va iniţializă cu valoarea 2 * 0, adică cu 0, deoarece la momentul respectiv câmpul *b* încă nu a apucat să primească o altă valoare;
- se revine în constructorul *SubClasa()*, se iniţializează *b* cu *3* şi *y* cu *2*3= 6*, cât returnează metoda *calcul()* din *SubClasa*.

Dacă în constructorul unei superclase se apelează o metodă care este redefinită în subclasă, la crearea unui obiect al subclasei există riscul ca metoda respectivă să refere câmpuri neiniţializate încă la momentul apelului.

Temă

1. Problema propusă încearcă să dea o mână de ajutor în gestionarea produselor unei firme care comercializează echipamente electronice. Fiecare echipament este înregistrat cu o *denumire*, un număr de inventar *nr_inv*, are un preţ *pret* şi este plasat într-o anumită zonă din magazie *zona_mag*. Orice echipament poate fi într-una din situaţiile:

- *achiziţionat* (intrat în magazie);
- *expus* (expus în magazin);
- *vândut* (transportat şi instalat la client).

Firma comercializează mai multe tipuri de echipamente. Toate echipamentele care folosesc hârtia drept consumabil sunt caracterizate de numărul de pagini scrise pe minut *ppm*. Imprimantele sunt caracterizate de *rezoluţie* (număr de puncte per inch *dpi*) şi număr de pagini/cartuş *p_car*. Unei imprimante i se poate seta *modul de scriere*:

- *tipărireColor* (selectere a modului color de tipărire);
- *tipărireAlbNegru* (selectere a modului alb-negru de tipărire).

Copiatoarele sunt caracterizate de numărul de pagini/toner *p_ton*. Se poate seta formatul de copiere:

- *setFormatA4* (setare a formatului A4);
- *setFormatA3* (setare a formatului A3).

Sistemele de calcul au un monitor de un anumit tip *tip_mon*, un procesor de o anumită viteză *vit_proc*, o capacitate a HDD *c_hdd* şi li se poate instala unul din *sistemele de operare*:

- *instalWin* (instalarea unei variante de Windows);
- *instalLinux* (instalarea unei variante de Linux).

Metodele de mai sus vor seta parametrii corespunzători.

Să se realizeze ierarhia de clase corespunzătoare modelului prezentat; Să se creeze UN SINGUR VECTOR DE OBIECTE în care să fie preluate datele din fişierul de intrare *electronice.txt.* Se va dezvolta un meniu care va oferi următoarele facilităţi:

- Afişarea imprimantelor
- Afişarea copiatoarelor
- Afişarea sistemelor de calcul
- Modificarea stării în care se află un echipament
- Setarea unui anumit mod de scriere pentru o imprimantă
- Setarea unui format de copiere pentru copiatoare
- Instalarea unui anumit sistem de operare pe un sistem de calcul
- Afişarea echipamentelor vândute
- Să se realizeze două metode statice pentru serializarea / deserializarea colecției de obiecte în fișierul echip.bin