Lecţia 5

Relaţia de moştenire

Între obiectele lumii care ne înconjoară există de multe ori anumite relaţii. Spre exemplu, putem spune despre un obiect autovehicul că are ca şi parte componentă un obiect motor. Pe de altă parte, putem spune că motoarele diesel sunt un fel mai special de motoare. Din exemplul secund derivă cea mai importantă relaţie ce poate exista între două clase de obiecte: *relaţia de moştenire*. Practic, relaţia de moştenire reprezintă *inima* programării orientate pe obiecte.

5.1 Ierarhizarea

La fel ca şi noţiunile de abstractizare şi încapsulare, ierarhizarea este un concept fundamental în programarea orientată pe obiecte. După cum am învăţat în prima lecţie, rolul procesului de abstractizare (cel care conduce la obţinerea unei abstracţiuni) este de a identifica şi separa, dintr-un punct de vedere dat, ceea ce este important de ştiut despre un obiect de ceea ce nu este important. Tot în prima lecţie am văzut că rolul mecanismului de încapsulare este de a permite ascunderea a ceea ce nu este important de ştiut despre un obiect. După cum se poate observa, abstractizarea şi încapsularea tind să micşoreze cantitatea de informaţie disponibilă utilizatorului unei abstracţiuni. O cantitate mai mică de informaţie conduce la o înţelegere mai uşoară a respectivei abstracţiuni. Dar ce se întâmplă dacă există un număr foarte mare de abstracţiuni?

Într-o astfel de situaţie, des întâlnită în cadrul dezvoltării sistemelor software de mari dimensiuni, simplificarea înţelegerii problemei de rezolvat se poate realiza prin ordonarea acestor abstracţiuni formându-se astfel *ierarhii* de abstracţiuni.

Definiţie 6 *O ierarhie este o clasificare sau o ordonare a abstracţiunilor.*

Este important de menţionat că ordonarea abstracţiunilor nu este una artificială. Între abstracţiuni există de multe ori implicit anumite relaţii. Spre exemplu, un motor este parte componentă a unei maşini. Într-o astfel de situaţie vorbim de o relaţie de tip *part*

of. Ca un alt exemplu, medicii cardiologi sunt un fel mai special de medici. Într-o astfel de situaţie vorbim de o relaţie de tip *is a* între clase de obiecte. În cadrul programării orientate pe obiecte, aceste două tipuri de relaţii stau la baza aşa numitelor *ierarhii de obiecte*, respectiv *ierarhii de clase*. În continuare vom discuta despre aceste două tipuri de ierarhii insistând asupra ierarhiilor de clase.

Imaginaţi-vă că sunteţi într-un hipermarket şi vreţi să cumpăraţi un anumit tip de anvelopă de maşină. Este absolut logic să vă îndreptaţi spre raionul denumit "Autovehicule". Motivul? Anvelopa este *parte componentă* a unei maşini şi implicit trebuie să fie parte a raionului asociat acestora. Ar fi destul de greu să găsiţi o anvelopă dacă aceasta ar fi plasată pe un raft cu produse lactate din cadrul raionului "Produse alimentare". Odată ajunşi la raionul "Autovehicule" veţi căuta raftul cu anvelope. Acolo veţi găsi o sumedenie de tipuri de anvelope de maşină, printre care şi tipul dorit de voi. Toate au fost puse pe acelaşi raft pentru că fiecare este în cele din urmă *un fel de anvelopă*. Dacă ele ar fi fost împrăştiate prin tot raionul "Autovehicule" ar fi fost mult mai complicat să găsiţi exact tipul de anvelopă dorit de voi. Acesta este numai un exemplu în care se arată cum relaţiile de tip *part of* şi *is a* pot conduce la o înţelegere mai uşoară a unei probleme, în acest caz organizarea produselor într-un hipermarket.

5.1.1 Ierarhia de obiecte. Relaţia de agregare

Ierarhia de obiecte este o ierarhie de tip întreg/parte. Să considerăm un obiect dintr-o astfel de ierarhie. Pe nivelul ierarhic imediat superior acestui obiect se găseşte obiectul din care el face parte. Pe nivelul ierarhic imediat inferior se găsesc obiectele ce sunt părţi ale sale.

Este simplu de observat că o astfel de ierarhie descrie relaţiile de tip *part of* dintre obiecte. În termeni aferenţi programării orientate pe obiecte o astfel de relaţie se numeşte *relaţie de agregare*.

În porţiunea de cod de mai jos se poate vedea cum este transpusă o astfel de relaţie în cod sursă Java. În acest exemplu un obiect maşină agregă un obiect motor. Figura 5.1 descrie modul de reprezentare UML a relaţiei de agregare dată ca exemplu, într-o diagramă de clase. Este interesant de observat că, deşi relaţia se reprezintă ca o relaţie între clase, agregarea se referă la obiecte (adică, fiecare *obiect Masina* are un *obiect Motor*).

```
class Masina {

    //Aceasta variabila contine o referinta la un obiect motor
    private Motor m;
```

```
    //Orice masina va trebui sa aiba un motor; semantic, n
    //nu ar trebui sa fie niciodata null
    public Masina(Motor n) {
        ...
        this.m = n;
        ...
    }

    //Elemente specifice unui obiect masina

}

class Motor {

    //Elemente specifice unui obiect motor

}
```

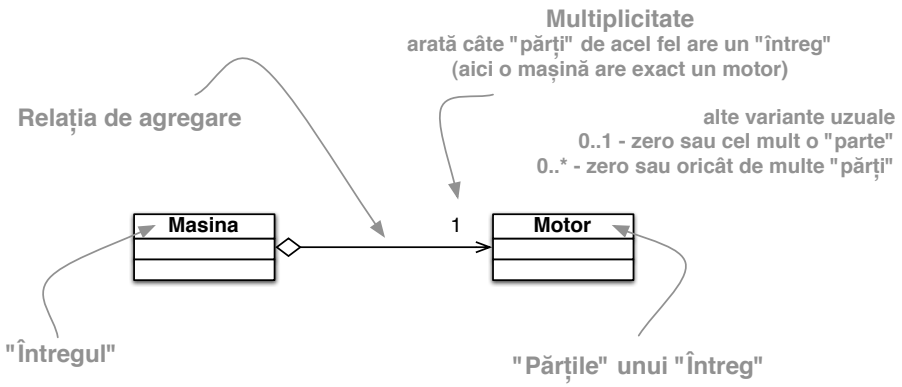

Figura 5.1: Reprezentarea UML a relaţiei de agregare.

Rezolvaţi Cum aţi implementa în cod sursă Java o relaţie de agregare în care un întreg poate avea 0 sau oricât de multe părţi (multiplicitate 0..*) ?

Să considerăm exemplul de mai jos. Reflectă această porţiune de cod o relaţie de agregare între un obiect maşină şi un obiect motor? Răspunsul corect este nu, pentru că din cod nu reiese că fiecare instanţă a clasei *Masina* are ca şi parte a sa o instanţă a clasei *Motor* (variabila m nu este câmp al clasei *Masina*). Este drept că în momentul execuţiei constructorului clasei

Masina se crează o instanţă a clasei *Motor* dar acest lucru denotă o altfel de relaţie între clase denumită *dependenţă*. Despre aceasta relaţie nu vom vorbi însă acum.

```
class Masina {

    public Masina() {
        Motor m;
        //Avem nevoie de un obiect Motor pentru a efectua anumite operatii
        //de initializare a unui obiect Masina. Dupa terminarea
        //constructorului nu mai e nevoie de acest obiect.
        m = new Motor();
        ...
    }
    //Elemente specifice unui obiect masina
}
```

5.1.2 Ierarhia de clase. Relaţia de moştenire

Ierarhia de clase este o ierarhie de tip generalizare/specializare. Să considerăm o clasă B care face parte dintr-o astfel de ierarhie. Pe nivelul ierarhic imediat superior se găseşte o clasă A care defineşte o abstracţiune mai generală decât abstracţiunea definită de clasa B. Cu alte cuvinte, clasa B defineşte un set de obiecte mai speciale inclus în setul de obiecte definite de clasa A. Prin urmare, putem spune că *B este un fel de A*.

După cum se poate observa, ierarhia de clase este generată de relaţiile de tip *is a* dintre clasele de obiecte, această relaţie numindu-se relaţie de moştenire. Într-o astfel de relaţie clasa A se numeşte *superclasă* a clasei B, iar B se numeşte *subclasă* a clasei A.

Toată lumea ştie că "pisica este un fel de felină". Trebuie să observăm că afirmaţia este una generală în sensul că "toate pisicile sunt feline". Ca urmare, afirmaţia se referă la clase de obiecte şi nu la un anumit obiect (nu se referă doar la o pisică particulară). Rezultatul este că între clasa pisicilor şi cea a felinelor există o relaţie de moştenire în care *Pisica* este subclasă a clasei *Felina* iar *Felina* este superclasă a clasei *Pisica*. În Figura 5.2 se exemplifică modul de reprezentare UML a relaţiei de moştenire între două clase.

După cum am spus încă de la începutul acestei lecţii, relaţia de moştenire este inima programării orientate pe obiecte. Este normal să apară întrebarea: de ce? Ei bine, limbajele de programare orientate pe obiecte, pe lângă faptul că permit programatorului să marcheze explicit relaţia de moştenire dintre două clase, mai oferă următoarele facilităţi:

- o subclasă preia (moşteneşte) reprezentarea internă (datele) şi comportamentul (metodele) de la superclasa sa.

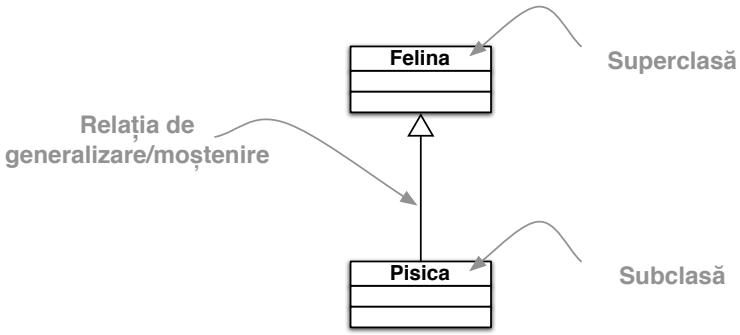

Figura 5.2: Reprezentarea UML a relaţiei de moştenire.

- un obiect instanţă a unei subclase poate fi utilizat în locul unei instanţe a superclasei sale.
- legarea dinamică a apelurilor metodelor.

În această lecţie ne vom rezuma exclusiv la prezentarea primelor două facilităţi, cunoscute şi sub numele de moştenire de clasă, respectiv moştenire de tip. Legarea dinamică va fi tratată în lecţia următoare.

5.2 Definiţia programării orientate pe obiecte

Relaţia de moştenire reprezintă elementul fundamental care distinge programarea orientată pe obiecte de programarea structurată. Acum că am descris relaţia de moştenire putem da o definiţie completă a programării orientată pe obiecte.

Definiţie 7 *Programarea orientată pe obiecte este o metodă de implementare a programelor în care acestea sunt organizate ca şi colecţii de obiecte care cooperează între ele, fiecare obiect reprezentând instanţa unei clase,* **fiecare clasă fiind membra unei ierarhii de clase ce sunt unite prin relaţii de moştenire.**

5.3 Declararea relaţiei de moştenire în Java

Exprimarea relaţiei de moştenire dintre o subclasă şi superclasa sa se realizează în Java utilizând cuvântul cheie *extends*.

```
class nume_subclasa extends nume_superclasa {
    // definirea elementelor specifice subclasei
}
```

Deşi această construcţie Java exprimă atât moştenirea de clasă cât şi moştenirea de tip între cele două clase, vom trata separat cele două noţiuni pentru a înţelege mai bine distincţia dintre ele.

5.4 Moştenirea de clasă în Java

Moştenirea de clasă este o facilitate a limbajelor de programare orientate pe obiecte care permite să definim implementarea unui obiect în termenii implementării altui obiect. Mai exact, o subclasă preia sau moşteneşte reprezentarea internă (datele) şi comportamentul (metodele) de la superclasa sa.

După cum se poate observa, această facilitate permite reutilizarea de cod. În contextul relaţiei de moştenire, dacă spunem că "o clasă B este un fel de clasă A" atunci se înţelege că orice "ştie să facă A ştie să facă şi B". Ca urmare, întregul cod sursă al clasei A ar trebui copiat în codul sursă al clasei B, lucru ce ar conduce la o creştere artificială a dimensiunii programului. Ei bine, prin moştenirea de clasă, această problemă e eliminată, subclasa moştenind implicit codul de la superclasa ei. Acest lucru permite programatorului care scrie clasa B să se concentreze exclusiv asupra elementelor specifice clasei B, asupra a ceea ce "ştie să facă clasa B în plus faţă de A".

5.4.1 Vizibilitatea membrilor moşteniţi. Specificatorul de access protected

Într-o lucrare anterioară am văzut că drepturile de acces la membrii unei clase pot fi menţionate explicit prin specificatori de access. Tot acolo am văzut care sunt regulile de vizibilitate impuse de specificatorii *public* şi *private*. În continuare vom extinde aceste reguli în contextul moştenirii de clasă. Reamintim că drepturile de acces trebuie discutate atât din perspectiva interiorului unei clase cât şi din perspectiva exteriorului (clienţilor) ei.

- În interiorul unei subclase pot fi referiţi doar acei membri moşteniţi de la superclasă a căror declaraţie a fost precedată de specificatorii de acces *public* sau *protected*. Accesul la membrii declaraţi *private* nu este permis deşi ei fac parte din instanţele subclasei.
- În general, clienţii unei subclase pot referi doar acei membri moşteniţi de la superclasă a căror declaraţie a fost precedată de specificatorii de access *public*.
- În general, clienţii unei clase nu pot accesa membrii clasei ce sunt declaraţi ca fiind *protected*.
- Dacă o subclasă este client pentru o instanţă a superclasei sale (de exemplu o metodă specifică subclasei primeşte ca argument o instanţă a superclasei sale) drepturile la membrii acelei instanţe sunt aceleaşi ca pentru un client obişnuit.

Sfat

În anumite condiţii, Java permite unui client al unei subclase să acceseze şi membrii moşteniţi declaraţi *protected*. Recomandăm evitarea acestei practici deoarece ea contravine definirii teoretice a specificatorului *protected*. În alte limbaje de programare obiectuale (de exemplu C++), accesul la membrii protected e permis doar în condiţiile menţionate mai sus.

Aceste reguli de vizibilitate sunt exemplificate în porţiunea de cod de mai jos. Se poate observa că din perspectiva unui client nu se face distincţie între membrii moşteniţi de o clasă şi cei specifici ei.

```
class SuperClasa {
    public int super_a;
    private int super_b;
    protected int super_c;
}

class SubClasa extends SuperClasa {

    public void metoda(SuperClasa x) {
        super_a = 1; //Corect
        super_b = 2; //Eroare de compilare
        super_c = 3; //Corect

        x.super_a = 1; //Corect
        x.super_b = 2; //Eroare de compilare
        x.super_c = 3; //Corect in anumite conditii(clasele sunt in acelasi
                       //pachet). Incercati sa evitati.
    }
}

class Client {

    public void metoda() {

        SuperClasa sp = new SuperClasa();
        SubClasa sb = new SubClasa();

        sp.super_a = 1; //Corect
        sp.super_b = 2; //Eroare de compilare
        sp.super_c = 3; //Corect in anumite conditii

        sb.super_a = 1; //Corect
        sb.super_b = 2; //Eroare de compilare
        sp.super_c = 3; //Corect in anumite conditii
    }
}
```

În UML, vizibilitatea membrilor protected se marchează cu simbolul #. Figura 5.3 exemplifică modul de reprezentarea a clasei *SuperClasa* din exemplul anterior.

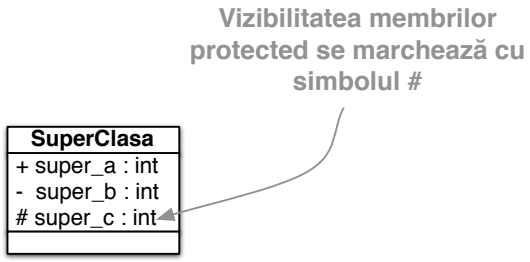

Figura 5.3: VIZIBILITATEA MEMBRILOR PROTECTED ÎN UML.

5.4.2 Cuvântul cheie super

Să considerăm exemplul de mai jos. Care câmp denumit *a* va fi iniţializat cu valoarea 1: cel moştenit sau cel privat?

```
class SuperClasa {
    protected int a;
}

class SubClasa extends SuperClasa {

    private int a;

    public void metoda() {
        this.a = 1;
    }
}
```

Standardul Java prevede ca în astfel de situaţii să se acceseze câmpul *a* local clasei *SubClasa*. Dacă dorim să accesăm câmpul *a* moştenit vom proceda ca mai jos, făcând uz de cuvântul cheie *super*. Acesta trebuie văzut ca o referinţă la "bucata" moştenită a obiectului apelat.

```
class SuperClasa {
    protected int a;
}
```

```
class SubClasa extends SuperClasa {

    private int a;

    public void metoda() {
        super.a = 1;
    }
}
```

5.4.3 Constructorii în contextul moştenirii de clasă

Într-o lucrare anterioară aţi învăţat că la crearea unui obiect trebuie specificat un constructor al clasei care se instanţiază, având rolul de a iniţializa într-un anumit mod obiectul creat. Prin urmare, la instanţierea unei subclase, trebuie să specificăm un constructor al respectivei subclase.

Pe de altă parte, în lucrarea de faţă am văzut că o subclasă moşteneşte câmpurile definite în superclasa sa. Mai mult, câmpurile moştenite ar putea fi private şi deci nu pot fi accesate din subclasă. Apare natural întrebarea: cum anume se iniţializează câmpurile moştenite de superclasă? Răspunsul vine la fel de natural: trebuie să apelăm undeva constructorul superclasei. Şi unde s-ar preta cel mai bine să apară acest apel? Evident, în interiorul constructorilor subclasei.

Standardul Java spune că prima instrucţiune din orice constructor al unei subclase trebuie să fie un apel la un constructor al superclasei sale.

Există o excepţie de la această regulă. Tot prima instrucţiune dintr-un constructor poate fi un apel la un alt constructor al aceleiaşi clase (e posibilă supraîncărcarea constructorilor). Într-o astfel de situaţie constructorul în cauză nu va mai apela deloc constructorul superclasei. Motivul e simplu: constructorul apelat va apela un constructor din superclasă.

Totuşi, sarcina introducerii acestui apel nu cade totdeauna în sarcina programatorului. Dacă superclasa are un constructor fără argumente (denumit şi constructor no-arg), compilatorul introduce singur un apel la acest constructor în toţi constructorii subclasei, cu excepţia cazului în care un constructor apelează alt constructor al subclasei. Acest lucru se întâmplă, chiar dacă subclasa nu are nici un constructor. După cum am învăţat într-o lecţie anterioară, dacă o clasă nu conţine nici un constructor compilatorul generează implicit un constructor no-arg pentru respectiva clasă. În cazul unei astfel de subclase, constructorul generat va conţine şi un apel la constructorul no-arg al superclasei sale.

În schimb, dacă superclasa are doar constructori cu argumente, programatorul trebuie să introducă explicit, în constructorii subclasei, un apel la unul din constructorii superclasei. În caz contrar se va genera o eroare la compilare deoarece compilatorul nu ştie care şi/sau cu ce parametri trebuie apelat constructorul superclasei. Acest lucru implică existenţa a cel puţin unui constructor în subclasă.

În continuare exemplificăm modul de apelare al unui constructor din superclasă. Se observă utilizarea cuvântului cheie *super* discutat în secţiunea anterioară.

```
class SuperClasa {

    private int x;

    public SuperClasa(int x) {
        this.x = x;
    }
}

class SubClasa extends SuperClasa {

    private int a;

    public SubClasa(int a,int x) {
        super(x);       //Apel la constructorul superclasei
        this.a = a;
    }

    public SubClasa(int a) {
        this(a,0);      //Apel la primul constructor.
                        //In acest constructor nu se mai poate apela
                        //constructorul superclasei.
    }
}
```

5.4.4 Exemplu de moştenire de clasă

Să considerăm o aplicaţie în care lucrăm cu numere complexe şi cu numere reale: trebuie să putem calcula modulul unui număr complex, trebuie să putem calcula modulul unui număr real, trebuie să putem afişa numerele reale şi complexe în forma *real + imaginar * i*, trebuie să putem compara două numere reale.

Pentru lucrul cu numerele complexe vom defini clasa de mai jos.

```
class NumarComplex {
```

```
    protected double re,im;

    public NumarComplex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public double modul() {
        return Math.sqrt( re * re + im * im );
    }

    public String toString() {
        return re + " + " + im + " * i";
    }
}
```

Pentru lucrul cu numere reale, trebuie să observăm că un număr real este un fel de număr complex. Mai exact, este un număr complex cu partea imaginară zero. Prin urmare, clasa *NumarReal* se defineşte astfel:

```
class NumarReal extends NumarComplex {

    public NumarReal(double re) {
        super(re,0);
    }

    public boolean maiMare(NumarReal a) {
        return re > a.re;
    }
}
```

Utilizând aceste două clase putem efectua operaţiile cerute în cerinţe.

```
class Client {

    public static void main(String argv[]) {

        NumarComplex a = new NumarComplex(1,1);
        System.out.println("Numarul este: " + a);
        System.out.println("Modulul sau este: " + a.modul());

        NumarReal c = new NumarReal(5);
        NumarReal d = new NumarReal(-6);
```

```
        System.out.println("Primul numar este: " + c);
        System.out.println("Modulul sau este: " + c.modul());
        System.out.println("Al doilea numar este: " + d);
        System.out.println("Modulul sau este: " + d.modul());
        System.out.println("E primul numar mai mare ca al doilea? - " +
               c.maiMare(d));
    }
}
```

Din acest exemplu se poate vedea cum *NumarReal* moşteneşte reprezentarea şi comportamentul clasei *NumarComplex*. Astfel, un număr real ştie să se tipărească şi să-şi calculeze modulul la fel ca un număr complex. În plus, un număr real ştie să se compare cu alt număr real.

Atenţie Codul de mai jos va genera o eroare de compilare. Acest lucru se întâmplă pentru că *a* este o referinţă la un obiect *NumarComplex* şi nu la *NumarReal*. Operaţia *maiMare* e specifică doar numerelor reale.

```
NumarComplex a = new NumarComplex(1, 0);
NumarReal b = new NumarReal(2);
a.maiMare(b);
```

5.5 Mosţenirea de tip în Java

Moştenirea de tip este o facilitate a limbajelor de programare orientate pe obiecte care permite să utilizăm (substituim) o instanţă a unei subclase în locul unei instanţe a superclasei sale. Să considerăm un exemplu.

```
class SuperClasa {...}

class SubClasa extends SuperClasa {...}

class Client {

    public void oMetoda() {
        ...
        SuperClasa a;
        SubClasa b = new SubClasa();
        a = b; //!!!//
        ...
    }
}
```

Datorită moştenirii de tip acest cod este corect, deoarece este permis să utilizăm un obiect *SubClasa* ca şi cum el ar fi o instanţă de tip *SuperClasa*. Este logic de ce e permis acest lucru: subclasa moşteneşte reprezentarea şi comportamentul de la superclasă. Prin urmare, tot ce ştie să facă superclasa ştie să facă şi subclasa. Aşadar, nu ne interesează dacă variabila *a* din exemplu referă un obiect *SubClasa*, pentru că sigur el va şti să se comporte şi ca o instantă din *SuperClasa*.

De ce se numeşte această facilitate moştenire de tip? Totalitatea metodelor publice ale unei clase reprezintă interfaţa obiectului definit iar din prima lecţie ştim că interfaţa denotă tipul obiectului. În contextul relaţiei de moştenire, o subclasă moşteneşte totul de la superclasă, deci şi metodele publice sau altfel spus tipul (interfaţa) ei. Din acest motiv se vorbeşte de moştenire de tip. Pe de altă parte subclasa ar putea defini noi metode publice, extinzând astfel tipul moştenit. Astfel, se spune că subclasa defineşte un subtip al tipului superclasei. Tipul superclasei se mai numeşte şi supertip pentru tipul subclasei.

5.5.1 Exemplu de utilizare a moştenirii de tip

Să considerăm acelaşi exemplu ca la moştenirea de clasă şi să presupunem că dorim să putem aduna două numere complexe, un număr real cu un număr complex sau două numere reale. Datorită moştenirii de tip, această problemă se poate rezolva adăugând clasei *NumarComplex* o metodă.

```
class NumarComplex {

    protected double re,im;

    public NumarComplex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public NumarComplex adunare(NumarComplex a) {
        return new NumarComplex(re + a.re, im + a.im);
    }

    public double modul() {
        return Math.sqrt( re * re + im * im );
    }

    public String toString() {
        return re + " + " + im + " * i";
    }
}
```

Un obiect *NumarComplex* poate fi adunat cu un obiect *NumarComplex* folosind metoda *adunare*. Un *NumarComplex* poate fi adunat cu un *NumarReal* deoarece metoda *adunare* poate primi ca parametru şi o instanţă *NumarReal* datorită moştenirii de tip. Clasa *NumarReal* moşteneşte metoda *adunare* deci se poate aduna cu un *NumarComplex* sau cu un *NumarReal*. Prin urmare cerinţele problemei au fost satisfăcute. Mai jos dăm un exemplu de utilizare a operaţiei de adunare.

```
class Client {

    public static void main(String argv[]) {

        NumarComplex a = new NumarComplex(1,1);
        NumarReal b = new NumarReal(5);

        System.out.println("Suma este:" + a.adunare(b));
        //Se obtine aceeasi suma si astfel
        System.out.println("Suma este:" + b.adunare(a));
    }
}
```

5.5.2 Operatorii instanceof şi cast

Datorită moştenirii de tip, o referinţă declarată de un anumit tip poate referi obiecte de orice subtip al tipului respectiv. În anumite situaţii este necesar să ştim tipul concret al obiectului indicat de o referinţă. Acest lucru se poate realiza prin operatorul *instanceof*.

```
referinta_obiect instanceof nume_clasa
```

O astfel de expresie are valoarea *true* dacă *referinta_obiect* indică un obiect instanţă a clasei *nume_clasa* sau a unei clase ce moşteneşte *nume_clasa*. Altfel valoarea expresiei este *false*. Mai jos dăm un exemplu de utilizare a operatorului, folosind clasele definite în secţiunea anterioară.

```
class Client {

    public static void test(NumarComplex x) {

        if (x instanceof NumarReal)
            System.out.println("NumarReal");
        else
            System.out.println("NumarComplex");
    }
```

```
    public static void main(String argv[]) {

        NumarComplex a = new NumarComplex(1,1);
        NumarReal b = new NumarReal(5);
        test(a); //Se va tipari NumarComplex
        test(b); //Se va tipari NumarReal
    }
}
```

În acest exemplu, metoda *test* îşi dă seama dacă parametrul său indică un obiect *NumărReal* sau nu. Să presupunem acum că aceeaşi metode trebuie să afişeze "NumarReal mai mare ca 0" sau "NumarReal mai mic sau egal cu 0" dacă parametrul său referă un obiect *NumarReal*.

```
class Client {

    public static void test(NumarComplex x) {
        if (x instanceof NumarReal) {
            NumarReal tmp = new NumarReal(0);
            if(x.maiMare(tmp)) //EROARE!!!
                System.out.println("NumarReal mai mare ca 0");
            else
                System.out.println("NumarReal mai mic sau egal cu 0");
        } else
            System.out.println("NumarComplex");
    }
}
```

Exemplul de mai sus va produce o eroare de compilare, datorită faptului că parametrul x este de tip *NumarComplex* iar un număr complex nu defineşte operaţia *maiMare*, ea fiind specifică obiectelor *NumarReal*. Soluţia constă în utilizarea operatorului *cast*, înlocuind linia marcată cu eroare cu linia de mai jos.

```
if (((NumarReal)x).maiMare(tmp))
```

Atenţie Dacă o instrucţiune de genul *((NumarReal)x).maiMare(tmp)* ajunge să se execute într-o situaţie în care x nu referă o instanţă *NumarReal*, se va semnala o eroare şi programul se va opri. Evident, în exemplul dat nu se întâmplă acest lucru pentru că am utilizat operatorul *instanceof* pentru a fi siguri că x referă o instanţă *NumarReal*.

Nu abuzaţi de operatorii *instanceof* şi *cast* pentru că sunt periculoşi. Dacă e posibil nici nu-i utilizaţi. Motivul e destul de greu de înţeles acum aşa că va trebui să ne credeţi pe cuvânt. În esenţă, moştenirea de tip ne permite să tratăm UNIFORM toate obiectele ale căror clase au o superclasă comună. De exemplu, instanţele claselor *NumarReal* şi *NumarComplex* pot fi toate privite ca instanţe ale clasei *NumarComplex*. Tratarea lor uniformă în majoritatea codului sursă face programul mai simplu de înţeles. În momentul în care se folosesc abuziv operatorii *instanceof* şi *cast* pentru a determina tipul real al obiectului, nu mai poate fi vorba de o tratare UNIFORMĂ. Programul devine mai greu de înţeles pentru că fiecare tip de obiect e tratat într-un mod particular lui (deci neuniform). Gândiţi-vă ce se întâmplă când avem 10, 20 sau 100 de tipuri de obiecte diferite (nu doar două). Complexitatea va deveni uriaşă şi din păcate va fi doar vina programatorului care a refuzat utilizarea facilităţii de moştenire de tip a limbajului. Un astfel de program NU mai este orientat pe obiecte, chiar dacă e scris în Java!!!

5.6 Exerciţii

1. Rulaţi şi studiaţi programele date ca exemplu în Secţiunile 5.4.4, 5.5 şi 5.5.2.
2. Fie o clasă *Punct* care are două câmpuri private x şi y reprezentând coordonatele sale în plan. Clasa are un singur constructor cu doi parametri care permite iniţializarea coordonatelor unui obiect *Punct* la crearea sa. Clasa *PunctColorat* extinde (moşteneşte) clasa *Punct* şi mai conţine un câmp c reprezentând codul unei culori. Argumentaţi dacă este sau nu necesară existenţa unui constructor în clasa *PunctColorat* pentru ca să putem crea obiecte *PunctColorat* şi, dacă da, daţi un exemplu de posibil constructor pentru această clasă.
3. Adăugaţi clasei *NumarComplex* dată ca exemplu în Secţiunea 5.5 o metodă pentru înmulţirea a două numere *NumarComplex*. Apoi scrieţi un program care citeşte de la tastatură o matrice de dimensiuni NxM şi o matrice de dimensiuni MxP, ambele putând conţine atât numere reale cât şi numere complexe (la citirea fiecărui număr utilizatorul specifică dacă introduce un numar complex sau unul real). În continuare, programul înmulţeşte cele două matrice (făcând uz de metodele de adunare şi înmulţire care sunt deja disponibile) şi afişează rezultatul pe ecran. Înmulţirea trebuie realizată într-o metodă statică ce primeşte ca parametri matricele de înmulţit.
4. Dorim să modelăm printr-un program Java mai multe feluri de avioane care formează flota aeriană a unei ţări. Ştim că această ţară dispune de avioane de călători şi de avioane de luptă. Avioanele de călători sunt de mai multe feluri, şi anume Boeing şi Concorde. De asemenea, avioanele de luptă pot fi Mig-uri sau TomCat-uri (F14). Fiecare tip de avion va fi modelat printr-o clasă iar avioanele propriu-zise vor fi instanţe ale claselor respective.

   Fiecare avion poate să execute o anumită gamă de operaţii şi proceduri, după cum se specifică în continuare. Astfel, orice avion trebuie să conţină un membru *planeID* de

tip *String* şi o metodă *public String getPlaneID()* care să returneze valoarea acestui membru. Mai mult, orice avion trebuie să conţină un membru *totalEnginePower* de tip întreg şi o metodă *public int getTotalEnginePower()* care să returneze valoarea acestui membru. Deoarece fiecare avion trebuie să poată decola, zbura şi ateriza, este normal ca pentru fiecare avion să putem apela metodele *public void takeOff()*, *public void land()* şi *public void fly()*. Metoda *takeOff()* va produce pe ecran textul "PlaneID_Value - Initiating takeoff procedure - Starting engines - Accelerating down the runway - Taking off - Retracting gear - Takeoff complete". Metoda *fly()* va produce pe ecran textul "PlaneID_Value - Flying". Metoda *land()* va produce pe ecran textul "PlaneID_Value - Initiating landing procedure - Enabling airbrakes - Lowering gear - Contacting runway - Decelerating - Stopping engines - Landing complete".

Avioanele de călători şi numai acestea trebuie să conţină un membru *maxPassengers* de tip întreg şi o metodă *public int getMaxPassengers()* care să returneze valoarea acestui membru. Avioanele de călători de tip Concorde sunt supersonice, deci are sens să apelăm pentru un obiect de acest tip metodele *public void goSuperSonic()* şi *public void goSubSonic()* care vor produce pe ecran "PlaneID_Value - Supersonic mode activated", respectiv "PlaneID_Value - Supersonic mode deactivated".

Avioanele de luptă şi numai acestea au posibilitatea de a lansa rachete asupra diferitelor ţinte, de aceea pentru orice avion de luptă trebuie să putem apela metoda *public void launchMissile()* care va produce pe ecran urmatorul text "PlaneID_Value - Initiating missile launch procedure - Acquiring target - Launching missile - Breaking away - Missile launch complete". Avioanele Mig şi numai acestea au geometrie variabilă pentru zbor de mare viteză, respectiv pentru zbor normal. Clasa corespunzătoare trebuie să conţină metodele *public void highSpeedGeometry()* şi *public void normalGeometry()* care vor produce pe ecran "PlaneID_Value - High speed geometry selected", respectiv "PlaneID_Value - Normal geometry selected". Avioanele TomCat şi numai acestea au posibilitatea de realimentare în zbor, deci pentru astfel de avioane are sens să apelăm o metodă *public void refuel()* care va produce pe ecran "PlaneID_Value - Initiating refueling procedure - Locating refueller - Catching up - Refueling - Refueling complete".

Se cere:

- Implementaţi corespunzător clasele diferitelor feluri de avioane. Din cerinţe rezultă că o parte din funcţionalitate/date este comună tuturor sau mai multor feluri de avioane în timp ce o altă parte este specifică doar avioanelor de un anumit tip. Prin urmare, părţile comune vor trebui factorizate făcând uz de moştenirea de clasă.
- Într-o metodă main, declaraţi mai multe variabile referinţă. Obligatoriu, toate variabilele vor avea acelaşi tip declarat. Creaţi apoi mai multe avioane (cel

puţin unul de fiecare fel). Pentru a referi aceste obiecte folosiţi doar variabilele amintite anterior bazându-vă pe moştenirea de tip. În continuare apelaţi diferitele operaţii disponibile fiecărui avion/fel de avion.

- Desenaţi diagrama UML de clase pentru ierarhia de clase obţinută.

Bibliografie

1. Grady Booch, *Object-Oriented Analysis And Design With Applications*, Second Edition, Addison Wesley, 1997.
2. Martin Fowler. *UML Distilled, 3rd Edition.* Addison-Wesley, 2003.
3. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison Wesley, 1999.
4. Carmen De Sabata, Ciprian Chirilă, Călin Jebelean, *Laboratorul de Programare Orientată pe Obiecte*, Lucrarea 5 - Relaţia de moştenire - Aplicaţii, UPT 2002, variantă electronică.