Lucrarea 2
Componența unei clase. Crearea şi inițializarea obiectelor. Tablouri

Cuprins

Componența unei clase

În limbajele de programare orientate pe obiecte, clasele pot fi considerate ca fiind mecanisme prin care programatorul îşi construieşte propriile sale tipuri de date, pe care, apoi, le va folosi aproape la fel cum foloseşte tipurile predefinite. Fiind un tip de date, unul dintre rolurile fundamentale ale clasei este acela de a servi la declararea variabilelor. Valorile unui tip clasă se numesc obiecte sau instanțe ale clasei respective.

O clasă reprezintă descrierea unei mulțimi de obiecte care au aceeaşi structură şi acelaşi comportament. Prin urmare, o clasă va trebui să conțină definițiile datelor şi ale operațiilor ce caracterizează obiectele acelei clase. Datele definite într-o clasă se mai numesc date-membru, variabile-membru, atribute sau câmpuri, iar operațiile se mai numesc metode sau funcţii membru. Pentru a arăta felul în care se definesc membrii unei clase, va fi utilizat următorul exemplu:

```
class Punct{
        private int x;
        private int y;
        public void init(int xx, int yy) {
              x = xx; y = yy;
        }
        public void move(int dx, int dy) {
              x += dx; y += dy;
        }
        public int getX( ) { return x; }
        public int getY( ) { return y; }
}
```

Clasa modelează un punct şi are ca date-membru două variabile de tip întreg: *x* şi *y*, reprezentând coordonatele punctului, iar ca metode:

- funcția *init()*, inițializează coordonatele unui punct
- funcția *move()*, deplasează un punct pe o anumită distanță
- funcțiile *getX()* şi *getY()*, returnează coordonatele curente ale unui punct.

Crearea obiectelor

În continuare se va considera un mic program Java pentru a vedea cum se creează obiectele clasei *Punct* descrisă anterior:

```
class Punct{
        private int x;
        private int y;
        public void init(int xx, int yy) {
                x = xx; y = yy;
        }
        public void move(int dx, int dy) {
                x += dx; y += dy;
        }
        public int getX( ) { return x; }
        public int getY( ) { return y; }
}

class MainApp {
        public static void main(String[ ] args) {
                Punct p1 = new Punct( ); //se creează o instanță a clasei Punct
                Punct p2 = new Punct( ); // şi încă una
                p1.init (10,20); p2.init (30,40); //se apeleaza metodele init ale instantelor
                p1.move(5,5); p2.move(6,-2); //se apeleaza metodele move
                System.out.println("(x1,y1) = ("+p1.getX()+","+p1.getY( )+ ")"); //se afiseaza
                        //coordonatele curente ale primului punct
                System.out.println("(x2,y2) = ("+p2.getX()+","+ p2.getY( )+ ")"); //se afiseaza
                        //coordonatele curente ale celui de-al 2-lea punct
        }
}
```

În *Java* toate obiectele se creează în mod dinamic. Până nu se inițializează variabila respectivă cu adresa unei zone de memorie alocată pentru valori ale tipului indicat de pointer, practic nu se poate folosi variabila în nici un fel. În programul de mai sus se observă aplicarea operatorului de alocare dinamică *new*.

Modificatori de acces

Se poate observa că definițiile datelor şi ale funcțiilor din clasa *Punct* sunt prefixate de anumite cuvinte cheie *(public, private*) care se numesc modificatori de acces. Aceştia stabilesc drepturile de acces ale clienților la membrii unei clase. Când se discută despre drepturile de acces la membrii unei clase trebuie să se abordeze acest subiect din două perspective:

- *Interiorul clasei sau, mai concret, metodele clasei.* În cadrul metodelor unei clase există acces nerestrictiv la toți membrii, date sau funcții. De exemplu, în metodele clasei *Punct* se face referire la câmpurile *x* şi *y*. În interiorul clasei nu se foloseşte notația cu punct pentru a referi membrii, aceştia fiind pur si simplu accesați prin

numele lor. Când o metodă face referire la alţi membri ai clasei, de fapt sunt accesaţi membrii corespunzători ai obiectului receptor, indiferent care ar fi el. De exemplu, când se apelează metoda *init()* a obiectului referit de *p1*, are loc iniţializarea membrilor *x* şi *y* ai acelui obiect. În legătură cu accesul din interiorul unei clase, trebuie spus că absenţa restricţiilor se aplică şi dacă este vorba despre membrii altui obiect din aceeaşi clasă, dar diferit de cel receptor. De exemplu, dacă în clasa *Punct* am avea câte o metodă de calcul a distanţei pe verticală/orizontală dintre 2 puncte, unul fiind obiectul receptor, iar celălalt un obiect dat ca parametru, atunci am putea scrie:

```
class Punct{
        //. . .
        public int distV(Punct p) {
             return y - p.y;
        }
        public int distH(Punct p) {
              return x - p.x;
        }
        //. . .
}
```

Se observă că din interiorul metodelor *distV() / distH()* putem accesa liber membrii privaţi ai obiectului *p* dat ca parametru. La fel ar sta lucrurile şi dacă *p* ar fi o variabilă locală a unei metode din clasa *Punct*.

- *Exteriorul sau clienţii clasei.* Clienţii unei clase pot accesa doar acei membri care au ca modificator de acces cuvântul public. Membrii declaraţi cu modificatorul private *NU* sunt vizibili în afară, sunt ascunşi. Dacă s-ar încerca folosirea, în metoda *main()* din exemplul considerat, o referinţă de genul:

```
p1.x
```

compilatorul ar raporta o eroare. Structura unei clase, sau modul ei de reprezentare, care este dat de variabilele membru, de regulă se ascunde faţă de clienţi. Dacă este necesar ca aceştia să poată consulta valorile datelor membru, se va opta pentru definirea unor metode de genul *getValoare()*, iar nu pentru declararea ca publice a datelor respective.

Iniţializarea câmpurilor unui obiect. Constructori

În exemplul considerat anterior, după crearea obiectelor *p1* şi *p2*, s-a apelat pentru ele metoda *init()* care avea ca scop iniţializarea câmpurilor *x* şi *y* pentru cele două puncte. Acesta este un exemplu de iniţializare a câmpurilor unui obiect, însă are unele dezavantaje printre care şi acela că programatorul ar putea omite apelul metodei, caz în care câmpurile vor avea valori implicite. În acest sens, programatorul poate prevedea la definirea unei clase una sau mai multe metode speciale, numite constructori, care servesc tocmai la iniţializarea datelor unui obiect imediat ce el a fost creat. Exemplul următor, descris tot folosind clasa Punct, va reliefa acest lucru:

```
class Punct{
        //. . .
        public Punct(int xx, int yy) {
              x = xx; y = yy;
        }
        public Punct(Punct p) {
             x = p.x; y = p.y;
        }
        //. . .
}
```

Caracteristicile unui constructor sunt:

- este o metodă care are acelaşi nume cu clasa în care este definită
- nu are tip returnat (nici măcar void)
- se apelează în mod automat la crearea unui obiect

Astfel, în locul metodei *init()*, în funcția *main()* a clasei *MainApp*, se poate scrie:

```
class MainApp {
        public static void main(String[ ] arg) {
             Punct p1 = new Punct(10,20 ); //se creează o instanță a clasei Punct
             Punct p2 = new Punct(30,40 ); // şi încă una
        //. . .
        }
}
```

O clasă poate avea mai mulți constructori, datorită facilității de supraîncărcare a funcțiilor existentă în limbajul *Java*. Se oferă, astfel, posibilitatea de a defini, în acelaşi domeniu de vizibilitate, mai multe funcții care au acelaşi nume, dar parametrii diferiți ca tip şi număr. Al doilea constructor definit anterior poate fi apelat astfel:

```
        Punct p3 = new Punct(p1);
```

Dacă programatorul nu prevede nici un constructor, atunci compilatorul va prevedea clasa respectivă cu un constructor implicit de tip *no-arg*, al cărui corp de instrucțiuni este vid. Aşa au stat lucrurile în cazul primei definiții a clasei *Punct* când la crearea unui obiect a fost apelat constructorul implicit *Punct()*. De asemenea, pentru a putea crea obiecte ale claselor descrise, constructorii trebuie să aibă modificatorul de acces public. Pe lângă constructori, mai există şi alte metode de inițializare a câmpurilor unei clase:

- *Inițializatori la declarare, care pot fi impliciți sau expliciți*:

```
class Punct{
        private int x; //inițializator implicit
        private int y = 1; //inițializator explicit
        //se presupune că nu există constructori
}
//. . .
Punct p1 = new Punct(); //în acest caz p1.x = 0 si
p1.y=1
```

Iniţializatorii impliciţi depind de tipul câmpului respectiv: sunt 0 pentru tipurile numerice, *false* pentru tipul boolean şi *null* pentru referinţe la obiecte.

- *Blocuri de iniţializare*: sunt secvenţe de cod cuprinse între acolade, plasate imediat după declaraţia câmpului pe care îl iniţializează, şi care pot fi asimilate cu nişte constructori *no-arg*. Ele se folosesc atunci când valoarea de iniţializare a unui câmp nu este o constantă simplă, ci trebuie obţinută prin calcule. Ca exemplu, se va presupune că variabila *x* din clasa *Punct* trebuie iniţializată cu valoarea sumei primelor 6 numere *Fibonacci*:

```
class Punct{
        private int x=2; //iniţializator explicit
        { //bloc de iniţializare
        int a=1,b=1;
        for(int i=3;i <= 6;i++)
        {
              b+=a; a=b-a;
              x+=b;
        }
//. . .
}
```

Deosebirea esenţială dintre constructori şi celelalte tipuri de iniţializări este aceea că, în cazul constructorilor, se putea iniţializa fiecare instanţă cu valori diferite care se transmit ca parametri. Celelalte mecanisme de iniţializare presupun ca toate instanţele clasei respective să aibă aceleaşi valori. Dacă se doreşte crearea şi iniţializarea unui câmp cu comportare echivalentă unei constante din limbajul *C++*, atunci câmpul este marcat ca *final*, cuvânt cheie plasat între modificatorul de acces şi tip.

Membri statici ai claselor

Până acum, în clasa *Punct* s-au definit doar membri *non-statici*. Aceştia se mai numesc şi membri ai instanţelor. Limbajul *Java* permite definirea unei categorii speciale de membri, numiţi *statici* sau membri de clasă. Aceşti membri vor exista în exemplare unice pentru fiecare clasă, fiind accesaţi în comun de toate instanţele clasei respective. *Mai mult, membrii statici pot fi referiţi chiar şi fără a instanţia clasa, ei nedepinzând de obiecte*. Pentru a defini un membru static se utilizează cuvântul cheie *static*, plasat după modificatorul de acces:

```
        modificator_acces static tip_membru nume_membru;
```

Referirea unui membru static nu se va face prin intermediul numelui obiectului, ci prin intermediul numelui clasei:

```
        nume_clasa.nume_membru_static
```

Pentru exemplificare, se va modifica clasa Punct astfel încât să se poată calcula numărul de apeluri ale metodei *move()* pentru toate obiectele create într-un program. Pentru aceasta, se va defini o variabilă statică *contor* şi o metodă statică *getContor()*:

```
class Punct{
        private int x;
        private int y;
        private static int contor=0; //inițializator implicit
        public Punct(int xx,int yy ) { x=xx; y=yy; }
        public void move(int dx, int dy) { x+=xx; y+=yy;contor++;}
        public static int getContor() {return contor;}
        //. . .
}
//. . .
        //metoda main din clasa rădăcină
        Punct p1 = new Punct(10,20);
        Punct p2 = new Punct(15,13);
        p1.move(8,-2); p2.move(6,7);
        //. . .
        System.out.println(”S-au executat “+Punct.getContor()+”mutari.”);
```

În legătură cu membrii statici ai unei clase trebuie făcută următoarea observație: într-o metodă statică nu este permisă referirea simplă a unui membru *non-static*. Acest lucru se explică prin aceea că un membru non-static există doar în interiorul unui obiect, pe când membrii statici există independent de obiecte. Dacă în exemplul dat s-ar fi declarat variabila contor ca *non-statică*, atunci ea ar fi existat în atâtea exemplare, câte obiecte s-ar fi creat şi ar fi contorizat pentru fiecare obiect în parte numărul de apeluri ale metodei *move()*.

Inițializarea variabilelor

În Java, variabilele al căror tip este o clasă sunt reprezentate ca referințe (adrese) spre obiecte ale clasei. Pentru a putea inițializa o asemenea referință, trebuie generat un obiect nou sau folosită o referință la un obiect deja existent. Unei referințe îi poate fi atribuită valoarea specială *null*, care se traduce prin faptul că referința respectivă nu indică nici un obiect. Valoarea *null* nu este atribuită automat tuturor variabilelor referință la declararea lor, ci conform următoarei reguli: dacă referința este o dată membru a unei clase şi ea nu este inițializată în nici un fel, la crearea unui obiect al clasei respective referința va primi implicit valoarea *null*. Dacă referința este o variabilă locală a unei metode, inițializarea implicită nu mai funcționează. De aceea, se recomandă ca programatorul să realizeze ÎNTOTDEAUNA o inițializare explicită a variabilelor.

Simbolul *this*

Simbolul *this* este o referință care poate fi utilizată doar în cadrul funcțiilor membru non-statice ale unei clase. Din punctul de vedere al unei funcții membru, acesta este referința spre obiectul receptor, „posesor“ al acelei funcții. Se poate spune că *this* reprezintă „conştiința de sine“ a unui obiect, în sensul că obiectul îşi cunoaşte adresa la care este localizat în memorie. Practic, orice referire a unei date membru nonstatice *v* în interiorul unei metode, poate fi considerată ca echivalentă cu *this.v*. De exemplu, metoda *move()* din clasa *Punct*, poate fi scrisă sub forma:

```
class Punct {
        private int x;
        private int y;
        //. . .
        public void move(int dx, int dy) {
            this.x += dx;
            this.y += dy;
        }
        //. . .
}
```

Referința *this* este folosită explicit în cazul în care ar putea exista conflicte de nume cu datele membru ale unui obiect. De exemplu:

```
class Punct {
        private int x,y;
        //. . .
        public Punct(int x, int y){
              // parametrii constructorului au nume
              // identice cu cele ale datelor membru
              this.x = x;
              this.y = y;
        }
}
```

O altă situație când se foloseşte explicit referința *this* este atunci când referința la obiectul receptor trebuie transmisă ca parametru la apelul unei alte metode.

```
class Context {
        private int x;
        private Algoritm a;
        // obiectul Algoritm va executa anumite calcule pentru un obiect Context

        public Context(Algoritm a, int x) {
            this.a = a;
            this.x = x;
        }
        public int Calcul() {
            x = a.Calcul(this);
            return x;
        }
        public int getX() { return x; }
}

class Algoritm {
        public int Calcul(Context c) {
            return c.getX()*c.getX();
        }
}
```

În fine, simbolul *this* reprezintă mijlocul prin care poate fi apelat un constructor al unei clase din interiorul altui constructor al aceleiaşi clase.

```
class OClasa {
        private int x,y;
        public OClasa(int x, int y) {
              this.x = x;
              this.y = y;
        }
        public OClasa(OClasa p) {
              this(p.x, p.y); // se apelează constructorul cu 2 parametri int
        }
        //. . .
}
```

Apelul unui constructor din interiorul altui constructor al clasei în cauză NU înseamnă crearea unui nou obiect, ci, pentru obiectul receptor curent, se va executa codul constructorului apelat, la fel ca în cazul apelării unei funcţii-membru obişnuite. De asemenea, un constructor nu poate fi apelat cu ajutorul simbolului *this* DECÂT din interiorul altui constructor şi nu al altor metode. În felul acesta se asigură îndeplinirea condiţiei ca pentru un obiect iniţializarea datelor prin constructor să se execute O SINGURĂ dată. Referinţa *this* NU poate fi utilizată în interiorul unei funcţii-membru statice. Referinţa *this* nu poate fi modificată (nu poate să apară ca membru stâng într-o atribuire) şi nu poate să apară în afara corpului de instrucţiuni al metodelor non-statice ale unei clase sau în afara blocurilor de iniţializare asociate cu variabilele membru ale unei clase.

Colectorul de reziduuri *(Garbage Collector)*

Spre deosebire de limbaje precum *Pascal* sau *C*, unde programatorul trebuie să elibereze memoria ocupată de obiecte după ce acestea nu mai sunt folosite, în Java programatorul este scutit de această sarcină de care se ocupă o componentă a maşinii virtuale numită *Garbage Collector*. Principiul de lucru al acestuia este următorul: dacă spre un anumit obiect nu mai există nici o referinţă externă, în nici o funcţie activă, acel obiect devine candidat la eliminarea din memorie.

Pentru utilizator, acţiunea *Garbage Collector*-ului este transparentă. Există posibilitatea de a „surprinde“ momentul în care un anumit obiect este eliminat, deoarece *Garbage Collector*-ul nu acţionează „fără preaviz“. Astfel, dacă în clasa de care aparţine obiectul este definită o metodă numită finalize(), cu prototipul:

```
        protected void finalize() throws Throwable;
```

această metodă va fi executată chiar înainte ca obiectul să dispară. Dacă metoda este definită astfel încât să tipărească pe ecran un anumit mesaj, poate fi detectat momentul în care un obiect este eliminat din memorie.

Tablouri în Java

Limbajul Java oferă tipul tablou (tip predefinit), cu ajutorul căruia se pot crea colecţii de variabile de acelaşi tip. Componentele tabloului pot fi accesate cu ajutorul unui index. Acesta trebuie să aparţină unui tip întreg, iar numerotarea elementelor într-un tablou începe întotdeauna de la 0, la fel ca şi în C. Un tablou se declară astfel:

```
tip_element[ ] nume_tablou;
```

În Java, elementele unui tablou se alocă DINAMIC, specificându-se dimensiunea tabloului, iar numele unui tablou este considerat ca o REFERINŢĂ DE OBIECT: tip_element[ ]

```
nume_tablou = new tip_element [numar_elemente];
```

La crearea unui tablou se memorează dimensiunile sale, programatorul putând în orice moment să le cunoască apelând la atributului *length*:

```
static void oFunctie(int[ ] tabp) {
        //. . .
        for(int i=0; i < tabp.length; i++) {
              // tabp.length returnează dimensiunea tabloului
              // prelucrează tabp[ i ]
        }

        }
        public static void main(String[ ] arg) {
              int [ ] tab = new int[10];
              //. . .
              oFunctie(tab);
              //. . .
        }
}
```

Dacă elementele unui tablou sunt, la rândul lor, de tip clasă, elementele respective vor fi referinţe la obiecte ale acelei clase. La crearea tabloului se rezervă spaţiu DOAR pentru acele referinţe şi ele sunt iniţializate cu valoarea *null*.

Java acceptă şi declaraţii de tablouri bidimensionale:

```
tip_element[ ] [ ] nume_matrice;
```

Temă

1. Să se parcurgă exemplele prezentate în laborator şi să se testeze practic (acolo unde este cazul).

2. Funcţia $f(x) = ax^2 + bx + c$ are ca grafic o parabolă cu vârful de coordonate $\left(-\frac{b}{2a}; \frac{-b^2 + 4ac}{4a} \right)$.

Se cere să se definească o clasă *Parabola* ai cărei membri vor fi:

- 3 variabile de tip double care reprezintă coeficienţii a, b şi c
- un constructor cu 3 parametrii de tip double
- un constructor cu un parametru de tip Parabola
- o metodă de afişare a funcţiei sub forma: f(x) = a x^2 + b x + c
- o metodă pentru calculul coordonatelor vârfului
- o metodă statică ce primeşte ca parametri două parabole şi calculează coordonatele mijlocului segmentului de dreptă care uneşte vârfurile celor două parabole astfel: $x = \frac{x_1 + x_2}{2}$, $y = \frac{y_1 + y_2}{2}$, unde (x_1, y_1) sunt coordonatele vârfului primei parabole, iar (x_2, y_2) descriu vârful celei de a doua parabole. Pe lângă clasa Parabola, se va mai defini o clasă MainApp care va exemplifica utilizarea metodelor clasei Parabola.

2. Să se scrie un program care citeşte dintr-un fişier text informaţii referitoare la produsele dintr-un magazin. Pe fiecare linie se află: denumirea produsului, preţul (număr real) şi cantitatea (număr real). Cele trei elemente sunt separate prin caracterul ";" (zahăr;1.5;50). Programul va afişa, sub forma unui tabel, conţinutul fişierului şi va determina denumirea, preţul şi cantitatea produsului cu preţ minim, respectiv maxim pe care le va scrie într-un fişier destinaţie. Programul va afişa pe ecran produsele pentru care cantitatea este mai mică decât o valoare citită de la tastatură.

Se va defini o clasă *Produs* care conţine ca date membru cele trei elemente ce descriu un produs. Pentru a tipări informaţiile cerute se va prevedea o metodă *toString()* în clasa *Produs*. Variabilele membre din clasa *Produs* vor fi private iar clasa va fi prevăzută cu cel puţin un constructor. Pentru fiecare produs din fişier se va crea un obiect de tip *Produs*.