Lecţia 3

Transmiterea mesajelor

Când apelăm o operaţie pusă la dispoziţie de un obiect prin intermediul interfeţei sale spunem că transmitem obiectului un mesaj. Obiectul care primeşte mesajul se numeşte obiect *apelat* sau *receptor*. Această lecţie prezintă câteva aspecte legate de transmiterea mesajelor.

3.1 Supraîncărcarea metodelor

În Java o metodă poate fi declarată doar în interiorul unei clase. Ea este alcătuită din două componente numite *prototipul* metodei, respectiv *corpul* metodei. Prototipul unei metode este alcătuit din specificatori, tip returnat, nume, lista parametrilor formali şi clauza opţională *throws* despre care vom vorbi mai târziu. Corpul unei metode este format dintr-un bloc de instrucţiuni iar în anumite condiţii, după cum vom vedea mai târziu, acesta poate lipsi.

Fiecare metodă are o *semnătură*.

Definiţie 5 *Semnătura unei metode este alcătuită din numele metodei împreună cu numărul şi tipurile parametrilor formali din prototipul metodei.*

Atunci când într-o clasă definim două sau mai multe metode cu acelaşi nume spunem că le supraîncărcăm. Pentru a face posibilă distincţia la nivelul compilatorului între metodele supraîncărcate trebuie ca semnăturile respectivelor metode să difere fie prin numărul de parametri formali ai metodei fie prin tipurile acestora.

Atunci când cineva spune că tipăreşte ceva, putem trage concluzia că o anumită informaţie va fi tipărită pe un suport fizic. Suportul fizic pe care va fi tipărită informaţia, dacă acesta prezintă interes, poate fi dedus din contextul în care se află persoana care realizează tipărirea: un programator va tipări pe ecran, un ziarist va tipări într-un ziar, etc. Aşa stau lu-

crurile şi cu o metodă supraîncărcată. Spre exemplu, e suficient să ştim că metoda *System.out.println(...)* tipăreşte pe ecran informaţia pe care o transmitem acesteia prin intermediul parametrilor. Dintr-un apel concret al acesteia putem afla exact şi ce tip de informaţie se va tipări (un întreg, un boolean).

Atenţie Metodele de mai jos sunt supraîncărcate, deşi la prima vedere au aceeaşi listă de parametri. În acest caz distincţia dinte ele se face în funcţie de ordinea tipurilor parametrilor actuali din cadrul apelului metodei.

```
void tipareste(String sir, boolean b) {
    if(b) System.out.println("String: " + sir);
}

void tipareste(boolean b, String sir) {
    if(b) System.out.println("String: " + sir);
}
```

Atenţie În literatură *supraîncărcarea* metodelor este cunoscută sub numele de *overloading*.

3.1.1 Supraîncărcarea constructorilor

Constructorii sunt în cele din urmă un fel de metode şi, prin urmare, pot fi supraîncărcaţi. Supraîncărcarea constructorilor permite instanţierea unui obiect în mai multe moduri, pentru fiecare mod existând câte un constructor. Spre exemplu, clasa *Valoare* de mai jos are doi constructori şi aceast fapt permite instanţierea obiectelor de acest tip în două moduri.

```
class Valoare {

    private int valoare;

    public Valoare() {
        valoare = 0;
    }

    public Valoare(int v) {
        valoare = v;
    }

    public void seteazaValoare(int v) {
        valoare = v;
    }
```

```
    public void tipareste() {
        System.out.println("Valoarea mea: " + valoare);
    }
}
```

Metoda *seteazaValoare* din interiorul clasei *Valoare* se numeşte *metodă accesor* pentru că singurul ei scop e setarea unei valori.

Datorită existenţei celor doi constructori, ambele moduri de instanţiere a obiectelor de mai jos sunt corecte.

```
Valoare o1 = new Valoare();
Valoare o2 = new Valoare(2);
```

3.1.2 Supraîncărcarea şi tipurile de date primitive

O variabilă de tip primitiv poate fi automat convertită spre un tip primitiv mai larg. Această conversie poate produce confuzii atunci când e vorba despre identificarea exactă a unei metode supraîncărcate care se apelează.

```
class ParametriPrimitivi {

    public static void care(float f) {
        System.out.println("FLOAT " + f);
    }

    public static void care(double f) {
        System.out.println("DOUBLE " + f);
    }

    public static void main(String[] args) {
        care(3.14);
        care(5);
    }
}
```

Spre exemplu, rezultatele produse de execuţia funcţiilor de mai sus vor fi

```
DOUBLE 3.14 //3.14 este implicit un double iar pentru a fi tratat
            //ca un float el trebuie succedat de sufixul f
FLOAT 5.0
```

Dacă există o metodă supraîncărcată al cărei parametru formal are exact acelaşi tip ca şi parametrul actual, se apelează acea metodă. Dacă, în schimb, nu există nici o metodă cu proprietatea de mai sus, atunci se va încerca o conversie automată de *lărgire* a tipului primitiv şi se va apela metoda având parametrul formal cel mai apropiat ca tip, dar mai mare decăt cel al parametrului actual.

Având în vedere că un *int* poate fi convertit automat la oricare din tipurile de mai jos

- *int* - *long*, *float* sau *double*

şi că avem două metode *care*, una cu un parametru formal de tip *float*, alta cu un parametru formal de tip *double* şi că tipul *float* e *cel mai apropiat* de tipul *int*, se va apela metoda *care* cu parametrul formal de tip *float*.

Tipul cel mai apropiat de tipul *char* este tipul *int*.

O variabilă de tip primitiv NU poate fi automat convertită spre un tip primitiv mai mic, conversia spre un tip primitiv mai mic fiind posibilă doar prin intermediul conversiilor explicite.

3.2 Transmiterea parametrilor

Atunci când definim o metodă ce are parametri, parametrii acesteia pot fi văzuţi ca nişte variabile locale ce vor primi valori concrete doar atunci când se apelează metoda. Parametrii din cadrul prototipului unei metode se numesc *parametri formali*. Parametrii ce se pun în punctul de apel al unei metode se numesc *parametri actuali*.

Există două moduri de transmitere a parametrilor:

- **prin valoare** – parametii formali vor fi înlocuiţi cu valorile parametrilor actuali; după execuţia metodei valorile parametrilor actuali nu vor fi schimbate.

- **prin referinţă(adresă)** – parametii formali vor referi exact aceeaşi zonă de memorie ca şi cei actuali; după execuţia metodei valorile parametrilor actuali pot fi schimbate.

În Java parametrii se transmit prin **valoare**!!!

În metoda *main* a clasei *TipPrimitiv* de mai jos se ilustrează efectul transmiterii prin *valoare* a unei variabile de tip primitiv. Evident, valoarea parametrului actual va fi *15* şi după execuţia metodei apelate.

```
class TipPrimitiv {

    public static void trimitValoarePrimitiva(int p) {
        p = 10;
    }

    public static void main(String[] arg) {
        int v = 15;
        trimitValoarePrimitiva(v);
        System.out.println("Valoarea lui v: " + v); //v va fi 15
    }
}
```

Dacă efectul transmiterii unui parametru de tip primitiv este evident, efectul transmiterii unui parametru de tip referinţă poate să nu fie aşa de evident. În acest caz, la apel, metoda va primi referinţa unui obiect, ceea ce se transmite *prin valoare* fiind o referinţă şi nu obiectul indicat de referinţă!!!

Aceasta înseamnă că modificările aduse asupra obiectului referit de parametrul *o* din exemplul de mai jos vor fi vizibile şi în metoda *main*. Pe de altă parte, modificarea referinţei din cadrul metodei *trimitValoareReferinta2* nu este vizibilă în metoda *main* datorită trimiterii prin *valoare* a referinţei.

```
class TipReferinta {

    public static void trimitValoareReferinta1(Valoare o) {
        o.seteazaValoare(10);
    }

    public static void trimitValoareReferinta2(Valoare q) {
        q = new Valoare(50);
    }

    public static void main(String[] arg) {
        Valoare v = new Valoare(5);
        v.tiparire(); //Va afisa Valoarea mea: 5

        trimitValoareReferinta1(v);
        v.tiparire(); //Va afisa Valoarea mea: 10 deoarece
                      //obiectul indicat de s si-a modificat continutul

        trimitValoareReferinta2(v);
        v.tiparire(); //Va afisa Valoarea mea: 10
                      //din cauza transmiterii referintei prin valoare
        }
}
```

Dacă dorim ca unui parametru de tip primitiv din cadrul prototipului unei metode să nu i se poată atribui în interiorul metodei o altă valoare decât cea primită la apel, în cadrul prototipului acesteia trebuie să apară corespunzător modificatorul *final*. Dacă parametrul este o referinţă, chiar dacă este specificat ca fiind *final*, prin intermediul său se pot aduce modificări obiectului referit dar nu şi referinţei însăşi.

3.3 Cuvântul cheie this

Cuvântul cheie *this* este o referinţă spre obiectul receptor al unui mesaj. *this* poate să apară doar în interiorul metodelor nestatice ale unei clase precum şi în interiorul constructorilor.

```
public void seteazaValoare(int v) {
    valoare = v;
    //echivalent cu
    //this.valoare = v;
}
```

Există mai multe situaţii care impun folosirea lui *this*. În continuare se vor exemplifica câteva dintre ele.

- Conflicte de nume – cu ajutorul lui *this* se poate face distincţie între un atribut aferent obiectului receptor al unui mesaj şi un parametru formal al mesajului, atunci când atributul, respectiv parametrul formal au acelaşi nume.

  ```
  public void seteazaValoare(int valoare) {
      this.valoare = valoare;
  }
  ```

- O metodă trebuie să returneze ca rezultat o referinţă la obiectul ei receptor.
- Referinţa la obiectul receptor trebuie transmisă ca parametru la apelul unei metode.
- Apelul unui constructor din alt constructor – în clasa *Valoare* vrem să adaugăm un constructor care primeşte ca parametru o referinţă la un obiect de tip *Valoare* şi vrem ca obiectul instanţiat în acest mod să aibă atributul său *valoare* egal cu cel al obiectului referit de *v*.

  ```
  public Valoare(Valoare v) {
      this(v.valoare);
      //se apeleaza constructorul Valoare(int v)
  }
  ```

Atenție Apelul unui constructor din interiorul altui constructor trebuie să fie prima instrucţiune din cadrul constructorului apelant, în caz contrar compilatorul va semnala o eroare. Apelul unui constructor din interiorul altui constructor nu înseamnă crearea unui alt obiect ci doar apelarea codului celuilalt constructor (pentru iniţializări).

Atenție Un constructor nu poate fi apelat cu ajutorul lui *this* din interiorul altor metode, ci doar din interiorul unui constructor.

Atenție Referinţa *this* nu poate fi folosită în interiorul unei metode statice deoarece metoda statică aparţine unei clase şi, de obicei, aceasta nu se execută pentru un obiect.

Atenție Referinţa *this* la obiectul receptor al unui mesaj nu poate fi modificată, adică acesteia nu i se poate atribui o altă valoare. Deci obiectul receptor al unui mesaj nu poate fi modificat pe parcursul tratării mesajului.

3.4 Metodele clasei Object

În Java orice clasă pe care o definim se află într-o relaţie specială cu o clasă numită *Object*. Ca urmare a acestei relaţii, orice clasă definită are şi ea toate metodele clasei *Object*. În Tabelul 3.1 sunt prezentate câteva metode ale clasei *Object*.

Prototipul	Descriere
Object clone()	Crează şi returnează o referinţă la o clonă a obiectului receptor
boolean equals(Object obj)	Testează dacă obiectul referit de obj este egal cu cel receptor
void finalize()	Se apelează la distrugerea obiectului de către Colectorul de reziduuri (Garbage Collector)
int hashCode()	Returnează codul hash al obiectului receptor
String toString()	Returnează o reprezentare sub formă de String a obiectului receptor

Tabelul 3.1: Câteva metode ale clasei Object.

3.4.1 Compararea obiectelor

După cum am vazut în Tabelul 1.4, atunci când comparăm valorile a două variabile de tip primitiv, folosim operatorul relaţional ==.

```
class TestareIdentitate {

    public static void main(String[] arg) {

        Valoare v1 = new Valoare(10);
        Valoare v2 = new Valoare(10);

        if(v1 == v2)
            System.out.println("Obiectele sunt egale");
        else
            System.out.println("Obiectele NU sunt egale");
    }
}
```

În Java operatorul relaţional == poate fi aplicat şi variabilelor de tip referinţă, ca în exemplul anterior. Poate surprinzător, rezultatul comparaţiei de mai sus este fals şi în consecinţă pe ecran se va afişa:

```
Obiectele NU sunt egale
```

De fapt, atunci când aplicăm operatorul == la două referinţe ceea ce comparăm este *identitatea fizică* a obiectelor referite, adică se verifică dacă cele două referinţe indică acelaşi obiect.

Dacă dorim să testăm *echivalenţa* dintre două obiecte, adică dacă două obiecte au conţinut identic, este bine să folosim metoda *equals* pe care orice clasă o are datorită relaţiei speciale dintre ea şi clasa *Object*. Dar această metodă *equals* trebuie *modificată* pentru fiecare clasă în parte astfel încât apelul acesteia să compare două obiecte din punct de vedere al *echivalenţei* (vom vedea mai târziu, de fapt, cum se numeşte această *modificare*). Fară *modificarea* de mai sus, metoda *equals* va testa *identitatea fizică* dintre obiectul receptor al metodei şi cel referit de parametrul metodei şi nu *echivalenţa* din punct de vedere al conţinutului acestora.

```
class Valoare {

    ...
    public boolean equals(Object o) {
        if(o instanceof Valoare)
            return (((Valoare)o).valoare == valoare);
        else
            return false;

    }
}
```

În interiorul metodei *equals* din cadrul clasei *Valoare* a trebuit să testăm dacă într-

adevăr s-a trimis o instanţă a clasei *Valoare* folosind operatorul *instanceof*. Dacă acest lucru s-a întâmplat, atunci rezultatul testării *echivalenţei* dintre cele două obiecte este dat de aplicarea operatorului == celor două atribute primitive.

Majoritatea claselor predefinite din Java oferă o implementare adecvată a metodei *equals* astfel încât aceasta să compare obiectele implicate din punt de vedere al *echivalenţei* şi nu al *identităţii*. Trebuie însă să consultaţi paginile de manual pentru a vedea ce înseamnă echivalenţa pentru acele clase.

Testarea *echivalenţei* dintre obiectele referite de *v1* şi *v2* se face în felul următor:

```
if(v1.equals(v2)) {
    System.out.println("Obiectele sunt egale dpdv al echivalentei");
} else {
    System.out.println("Obiectele NU sunt egale dpdv al echivalentei");
}
```

Se poate testa echivalenţa dintre două obiecte şi prin intermediul unei metode care nu se numeşte *equals* dar existenţa unei alte metode pentru efectuarea comparaţiei va reduce drastic înţelegerea programelor şi va duce la pierderea unor facilitaţi oferite de Java, facilităti despre care vom vorbi mai târziu.

3.4.2 Metoda toString

Metoda *toString* a clasei *Object*, ca şi celelalte metode ale acestei clase, este şi ea existentă în fiecare clasă pe care o definim. În consecinţă, fiecare obiect creat de noi are o metodă *toString* pe care o putem oricând apela în program.

```
//1 se apeleaza metoda supraincarcata println cu un parametru de tip String
System.out.println(v1.toString());
//2 se apeleaza metoda supraincarcata println cu un parametru de tip Object
System.out.println(v1);
```

După cum prezintă Tabelul 3.1, metoda returnează o reprezentare sub formă de *String* a obiectului receptor. Concret, efectul execuţiei primului apel de mai sus poate fi afişarea pe ecran a textului

```
Valoare@fd13b5
//pentru o alta executie, codul hash fd13b5 al obiectului v1 va fi altul
```

Reprezentarea implicită sub formă de şir de caractere este formată din numele clasei pe care o instanţiază obiectul receptor al metodei *toString* urmat de @ şi de codul hash

al obiectului. În continuare vom vedea care este rostul acestei metode.

Dacă ne uităm în documentaţia Java disponibilă la adresa *http://java.sun.com/j2se/1.5.0/docs/api/java/io/PrintStream.html* vom vedea că metoda *println* este supraîncărcată, existând printre prototipurile ei următoarele:

```
public void println(String x)
public void println(Object x)
```

Faptul că există o metodă *println* având un parametru de tip *String* va produce apelarea acesteia în primul caz. În al doilea caz, se va apela cea care are parametrul de tip *Object* datorită faptului că orice obiect instanţiat este într-o o relaţie specială cu clasa *Object*. Ceea ce face a doua metodă *println* nu este altceva decât afişarea pe ecran a şirului de caractere returnat de apelul metodei *x.toString()* şi, deci, se va tipări acelaşi şir de caractere ca şi în cazul primului apel.

Reprezentarea sub formă de *String* pe care o oferă implicit clasa *Object* nu este satisfăcătoare. De exemplu, pentru o instanţă a clasei *Valoare* reprezentarea sub formă de *String* ar putea fi

```
Valoarea mea este 10
```

Pentru ca reprezentarea sub formă de *String* a unei instanţe a clasei *Valoare* să fie cea de mai sus, metoda *toString* moştenită de la clasa *Object* trebuie *modificată* ca mai jos.

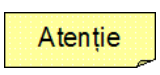
```
class Valoare {

    ...
    public String toString() {
        return "Valoarea mea este " + valoare;
    }
}
```

Dar oare care va fi efectul execuţiei codului de mai jos?

```
System.out.println("***" + v1);
```

În Java operatorul + este folosit şi pentru concatenări de şiruri de caractere. Pentru cazul de mai sus al doilea operand nu este altcineva decât reprezentarea sub formă de *String* a lui v1 iar, în consecinţă, pe ecran se va afişa

```
*** Valoarea mea este 10
```



Nu este bine să avem în clasa *Valoare* o metodă de genul:

```
public void afiseaza() {
    System.out.println("Valoarea mea este " + valoare);
}
```

3.4.3 Eliberarea memoriei ocupate de obiecte

Până acum am văzut cum se crează dinamic un obiect dar nu am spus nimic despre eliberarea memoriei ocupate de către acesta.

În Java, programatorul nu trebuie să elibereze explicit zone de memorie ocupate de obiectele instanţiate, acest lucru fiind realizat automat de către suportul de execuţie. Unul din cazurile în care un obiect devine automat candidat la ştergere (prin ştergere înţelegându-se *eliberarea zonei de memorie alocate pentru el la crearea sa*) este atunci când nu mai există nici o referinţă spre el. În exemplul de mai jos, primul obiect instanţiat va deveni candidat la ştergere deoarece v referă după a doua instrucţiune un alt obiect şi nu există o altă referinţă spre primul obiect creat. Este sarcina colectorului de reziduuri (garbage collector - GC) să elibereze memoria ocupată de primul obiect.

```
Valoare v;
v = new Valoare(5);
v = new Valoare(10);
```

Atenţie Nu este obligatoriu ca un obiect să fie şters din memorie imediat ce dispare ultima referinţă la el. Cert este că el devine candidat la ştergere iar momentul în care se întâmplă ştergerea depinde de gradul de ocupare al memoriei heap.

Atenţie Putem forţa ştergerea din memorie a tuturor obiectelor candidate pentru această operaţie invocând explicit colectorul de reziduuri *System.gc()*.

Atunci când colectorul de reziduuri eliberează zona de memorie ocupată de un obiect, el invocă metoda *finalize*:

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

Evident, această metodă trebuie *modificată* atunci când e nevoie să se efectueze anumite operaţii speciale la ştergerea unui obiect din memorie.

Atenţie E posibil ca un obiect să nu fie colectat de către GC pe întreg parcursul execuţiei unui program, caz în care metoda *finalize* nu va fi deloc apelată.

3.5 Exerciţii

1. Metodele de mai jos sunt supraîncărcate?

```
public void faCeva(int x) {...}
public int  faCeva(int x) {...}
```

2. O carte este caracterizată printr-un număr de pagini. Spunem că două cărţi sunt identice dacă acestea au acelaşi număr de pagini. Creaţi clasa *Carte* şi ataşaţi-i o metodă *potrivită* pentru compararea a două cărţi. Apelaţi metoda care realizează compararea a două cărţi într-o metodă *main*.
3. Un pătrat este caracterizat de latura sa. Scrieţi o clasă *Patrat* ce are doi constructori, un constructor fără nici un parametru care setează latura pătratului ca fiind *10* iar altul care setează latura cu o valoare egală cu cea a unui parametru transmis constructorului. Ataşaţi clasei o metodă *potrivită* pentru tipărirea unui pătrat sub forma *"Patrat" l "Aria" a*, unde *l* este valoarea laturii iar *a* este valoarea ariei pătratului. Creaţi într-o metodă *main* diferite obiecte de tip *Patrat* şi tipăriţi-le.
4. Creaţi o clasă *Piramida* ce are un atribut întreg *n*. Ataşaţi clasei o metodă *potrivită* pentru tipărirea unei piramide ca mai jos:

```
1 1 1 1
2 2 2
3 3
4 --> n
```

   Creaţi într-o metodă *main* diferite obiecte de tip *Piramida* şi tipăriţi-le.
5. Definiţi o clasă *Suma* cu metodele statice de mai jos:

```
// returneaza suma dintre a si b
a) public static int suma(int a, int b) ...
//returneaza suma dintre a, b si c
b) public static int suma(int a, int b, int c) ...
// returneaza suma dintre a, b, c si d
c) public static int suma(int a, int b, int c, int d) ...
```

   Implementaţi metodele astfel încât fiecare metodă să efectueze o singură adunare. Apelaţi-le dintr-o metodă *main*.

Bibliografie

1. Bruce Eckel. *Thinking in Java, 4th Edition*. Prentice-Hall, 2006. Capitolul Initialization & Cleanup.
2. *Java Language Specification*. http://java.sun.com/docs/books/jls/, Capitolul 8.4, Method Declarations.