Lecţia 10

Colecţii de obiecte

Presupunem că trebuie scrisă o aplicaţie care să gestioneze informaţii despre angajaţii unei companii dezvoltatoare de software. În companie există două feluri de angajaţi: contabili şi programatori. În acest context se pune problema modului în care se pot stoca informaţiile despre angajaţii companiei.

```
class Angajat {

    private String nume, prenume;
    private String departament;

    public Angajat(String nume, String prenume, String departament) {
        this.nume = nume;
        this.prenume = prenume;
        this.departament = departament;
    }

    public void schimbaDepartamentul(String departament) {
        this.departament = departament;
    }

    public String toString() {
        return "Nume:" + nume + " Departament:" + departament;
    }

    public boolean equals(Object o) {
        return (o instanceof Angajat) && ((Angajat)o).nume.equals(nume)
                                && ((Angajat)o).prenume.equals(prenume);
    }
    ...
}
```

Doi angajaţi se consideră identici din punct de vedere al conţinutului dacă aceştia au acelaşi nume, respectiv acelaşi prenume.

```
class Contabil extends Angajat {

    public Contabil(String nume, String prenume) {
        super(nume, prenume, "contabilitate");
    }
    ...
}

class Programator extends Angajat {

    public Programator(String nume, String prenume) {
        super(nume, prenume, "dezvoltare sisteme");
    }
    ...
}
```

10.1 Să ne amintim...tablourile

După cum am văzut într-una din lecţiile anterioare, o variantă pentru stocarea unei colecţii de obiecte de acelaşi tip o reprezintă tablourile.

```
Angajat[] a1 = new Angajat[2];

a1[0] = new Contabil("Popescu","Mircea");
a1[1] = new Programator("Ionescu","Mihai");

//sau
a1 = new Angajat[] { new Contabil("Popescu","Mircea"),
                     new Programator("Ionescu","Mihai")};
```

Spre deosebire de C sau C++ unde o metodă putea returna un pointer spre un tablou, în Java o metodă poate returna o referinţă la un obiect tablou, ca mai jos.

```
Angajat[] creazaAngajati() {
    Angajat[] angajati = ...
    ...
    return angajati;
}
```

În multe cazuri, după ce un tablou a fost creat, asupra sa se doresc a se efectua operaţii de genul căutare, tipărire, testarea egalităţii dintre două tablouri din punctul de vedere al conţinutului stocat, etc.

Dacă încercăm să "tipărim" un obiect tablou ca mai jos, vom observa că ceea ce se va tipări va fi reprezentarea implicită a fiecărui obiect sub formă de şir de caractere, adică numele clasei instanţiate precum şi codul hash al obiectului tipărit.

```
//Se va tipari LAngajat;@11b86e7
System.out.println(a1);
```

Totuşi, de obicei, prin tipărirea unui tablou se înţelege tipărirea tuturor elementelor existente în cadrul tabloului. Ei bine, această operaţie poate fi realizată apelând metoda statică *String toString(Object[] a)* a clasei *Arrays* din pachetul *java.util*.

```
//Se va tipari
//[Nume:Popescu Departament:contabilitate,
// Nume:Ionescu Departament:dezvoltare sisteme]
System.out.println(Arrays.toString(a1));
```

Dacă în clasa *Angajat* nu am fi suprascris metoda *toString*, în loc de afişarea de mai sus, pentru fiecare obiect angajat stocat în cadrul tabloului s-ar fi afişat numele clasei *Angajat* urmat de codul hash corespunzător fiecărui obiect. Acest lucru s-ar fi întâmplat deoarece metoda *toString* din clasa *Arrays* apelează pentru fiecare obiect conţinut metoda *toString* corespunzătoare.

O detaliere a unor metode existente in clasa *Arrays* se află în Tabelul 10.1 [1]. Referitor la prima metodă din cadrul Tabelului 10.1 este important de spus că două tablouri sunt considerate a fi egale din punct de vedere al conţinutului dacă ambele tablouri conţin acelaşi număr de elemente iar elementele conţinute sunt egale, mai mult, ele fiind stocate în aceeaşi ordine. Două obiecte referite de *o1* şi *o2* sunt egale dacă *(o1==null ? o2==null : o1.equals(o2))*.

În Secţiunea 3.4.2 am spus că nu este bine să avem într-o clasă o metodă de genul *public void afiseaza()*. Pe lângă faptul că absenţa metodei *toString()* ar necesita pentru fiecare mediu nou de afişare(fişier, casetă de dialog) introducerea unei noi metode care să afişeze obiectul receptor pe noul mediu, aceasta ar face imposibilă şi folosirea metodelor din clasa *Arrays* în scopul în care acestea au fost create.

Una dintre problemele principale existente atunci când utilizăm tablourile ca suport de stocare a colecţiilor de elemente este dimensiunea fixă a capacităţii de stocare. Atunci când dorim să modificăm dimensiunea unui tablou, trebuie să creem întâi un alt tablou iar apoi să copiem elementele din vechiul tablou în noul tablou. În acest scop putem folosi metoda *public static void arraycopy* a clasei *System*. Parametrii acestei metode

[1] În clasa Arrays aceste metode sunt supraîncărcate, ele existând şi pentru fiecare tip primitiv.

Prototipul	Descriere
boolean equals(Object[] a, Object[] a2)	Testează egalitatea dintre două tablouri din punct de vedere al conţinutului
String toString(Object[] a)	Returnează un şir de caractere ce conţine reprezentările sub formă de şiruri de caractere a tuturor obiectelor stocate în tabloul referit de *a*
void sort(Object[] a)	Sortează elementele tabloului referit de *a*. Pentru a putea fi sortate, toate elementele din tablou trebuie să fi implementat în prealabil interfaţa *Comparable*

Tabelul 10.1: Câteva metode statice definite de clasa Arrays.

sunt prezentaţi în detaliu la adresa
http://java.sun.com/j2se/1.5.0/docs/api/java/lang/System.html.

```
//numarul de angajati a crescut
Angajat[] a = new Angajat[4];
//copiem continutul vechiului tablou in noul tablou
System.arraycopy(a1,0,a,0,a1.length);
//a1 va referi noul tablou
a1 = a;
```

Până în acest moment tabloul de angajaţi nu s-a aflat într-o relaţie de agregare cu nici un alt obiect. Însă se pune problema de apartenenţă a colecţiei de angajaţi la un obiect. În acest caz, angajaţii aparţin unei companii.

```
class Companie {

    private String nume;
    private Angajat[] angajati;
    private int nrAngajati = 0;

    public Companie(String nume, int nrAngajati) {
        this.nume = nume;
        angajati = new Angajat[nrAngajati];
    }

    public void addAngajat(Angajat a) {
        if(nrAngajati<angajati.length)
```

```
            angajati[nrAngajati++]=a;
        else {
            //Marim numarul maxim de angajati al companiei
            Angajat[] ang = new Angajat[angajati.length+10];
            System.arraycopy(angajati,0,ang,0,angajati.length);
            angajati = ang;
            angajati[nrAngajati++]=a;
        }
    }
    ...
}
```

Ce s-ar întâmpla dacă, la un moment dat, ar trebui să se creeze un concurs între angajaţii tuturor companiilor producătoare de software? Este evidentă necesitatea stocării unei colecţii de angajaţi într-o altă clasă, posibil numită *Joc*.

```
class Joc {

    private Angajat[] angajati;
    private int nrParticipanti = 0;

    public Joc(int nrParticipanti) {
        angajati = new Angajat[nrParticipanti];
    }

    public void addAngajat(Angajat a) {
        if(nrParticipanti<angajati.length)
            angajati[nrParticipanti++]=a;
        else {
            //Marim numarul maxim de angajati participanti
            Angajat[] ang = new Angajat[angajati.length+10];
            System.arraycopy(angajati,0,ang,0,angajati.length);
            angajati = ang;
            angajati[nrParticipanti++]=a;
        }
    }
    ...
}
```

Se observă că între implementările celor două clase de mai sus, *Companie* respectiv *Joc*, există o secvenţă de *cod duplicat*. În Secţiunea 6.2 se spunea că e bine să scriem programe astfel încât acestea să nu conţină duplicare de cod. O variantă posibilă de eliminare a duplicării de cod ar fi crearea unei clase *ListaAngajati* a cărei funcţionalitate să se rezume strict la stocarea unei colecţii de angajaţi.

```
class ListaAngajati {

    private Angajat[] angajati;
    private int nrAngajati = 0;

    public ListaAngajati(int nr) {
        angajati = new Angajat[nr];
    }

    public void addAngajat(Angajat a) {
        if(nrAngajati<angajati.length)
            angajati[nrAngajati++]=a;
        else {
            //Marim numarul maxim de angajati participanti
            Angajat[] ang = new Angajat[angajati.length+10];
            System.arraycopy(angajati,0,ang,0,angajati.length);
            angajati = ang;
            angajati[nrAngajati++]=a;
            ...
        }
    }
   ...
}
```

În acest caz un obiect de tip *Companie* va agrega, în loc de un tablou de angajaţi ale cărui elemente să trebuiască a fi gestionate în interiorul clasei, un obiect *ListaAngajati* care se va ocupa de gestiunea angajaţilor.

```
class Companie {
    private String nume;
    private ListaAngajati lista;

    public Companie(String nume, int nrAngajati) {
        this.nume = nume;
        lista = new ListaAngajati(nrAngajati);
    }

    public void addAngajat(Angajat a) {
        lista.addAngajat(a);
    }
    ...
}
```

Conform cu cele precizate mai sus, de fiecare dată când e nevoie de o colecţie de obiecte trebuie să creăm mai întâi o clasă care să se ocupe de gestionarea(adăugarea, ştergerea, etc.) obiectelor din interiorul colecţiei. Din fericire, Java pune la dispoziţie un suport pentru lucrul cu colecţii de obiecte, acest lucru făcând inutilă crearea în acest scop a propriilor clase.

10.2 Suportul Java pentru lucrul cu colecţii de obiecte

Anterior am precizat că Java ne pune la dispoziţie un suport pentru lucrul cu colecţii de obiecte – în cadrul secţiunii de faţă se doreşte prezentarea unor aspecte legate de suportul menţionat.

10.2.1 O privire de ansamblu

Până acum am folosit din abundenţă termenul de *colecţie de obiecte* dar nu am dat o definiţie concretă a acestui termen. Acum e momentul să facem acest lucru.

Definiţie 3 *O colecţie este un grup de obiecte.*

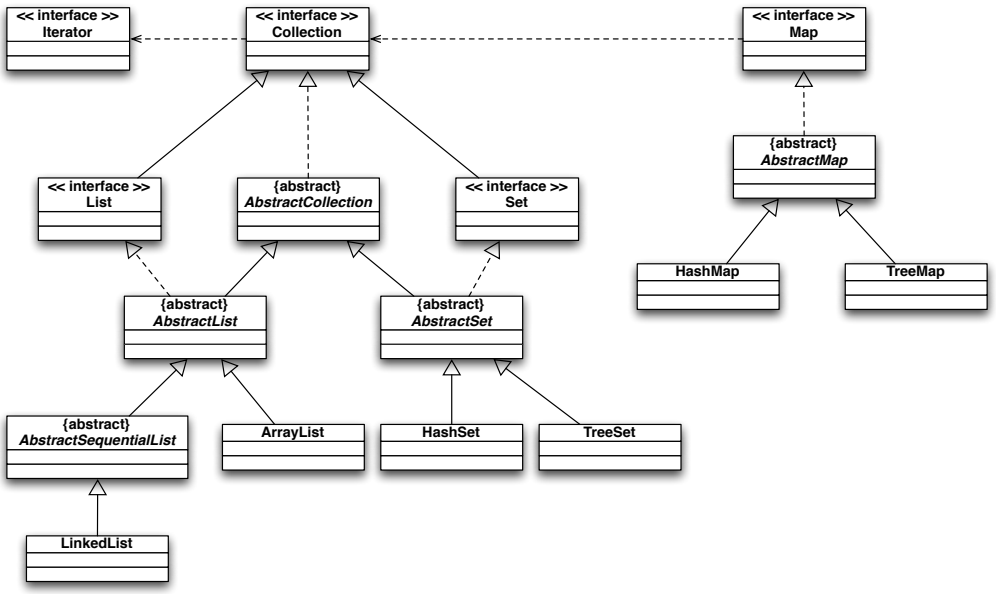


Figura 10.1: O vedere simplificată asupra sistemului de colecţii Java 1.4.

Clasele şi interfeţele pe care le oferă Java în vederea lucrului cu colecţii de obiecte se află în pachetul *java.util* şi, în consecinţă, pentru a le putea folosi uşor trebuie să folosim clauze import corespunzătoare, de exemplu ca mai jos:

```
import java.util.*;
```

În Figura 10.1 sunt reprezentate câteva interfeţe şi clase din acest pachet. Toate clasele concrete care au ca supertip interfaţa *Collection* implementează conceptul de colecţie de obiecte.

Toate clasele concrete care au ca supertip interfaţa *List* implementează conceptul de listă. O listă este o colecţie de obiecte în care fiecare obiect este reţinut într-o ordine bine precizată.

Toate clasele concrete care au ca supertip interfaţa *Set* implementează conceptul de mulţime. O mulţime este o colecţie de obiecte în care un obiect este reţinut o singură dată.

Toate clasele concrete care au ca supertip interfaţa *Map* implementează conceptul de dicţionar cheie-valoare.

10.2.2 Interfaţa Collection

Având în vedere că interfaţa *Collection* este implementată de ambele tipuri de colecţii, liste şi mulţimi, se impune detalierea în Tabelul 10.2 a metodelor existente în această interfaţă.

10.2.3 Liste

Se observă că în interiorul interfeţei *Collection* nu există metode care să întoarcă un element de pe o poziţie specificată. Acest lucru se datorează faptului că interfaţa este implementată şi de mulţimi iar în interiorul mulţimilor ordinea elementelor nu este neapărat ordinea în care elementele au fost adăugate, ea neavând nici o semificaţie. Interfaţa *List*, pe lângă metodele moştenite de la interfaţa *Collection*, mai are şi alte metode proprii, unele dintre ele fiind prezentate în Tabelul 10.3.

Printre clasele predefinite care implementează interfaţa *List* sunt *ArrayList* şi *LinkedList* iar în continuare vom prezenta câteva dintre caracteristicile acestor implementări.

Clasa *ArrayList* furnizează o implementare a interfeţei *List*, elementele propriu-zise ale liste fiind stocate de un tablou care este redimensionat. Această implementare faca ca accesul aleator al unui element să se facă foarte rapid. În schimb, ştergerea şi inserarea unui element în interiorul listei este o operaţie consumatoare de timp în raport cu implementarea *LinkedList*.

Clasa *LinkedList* furnizează o implementare a interfeţei *List*, elementele propriu-zise

Prototipul	Descriere
boolean add(Object o)	Asigură faptul că obiectul referit de *o* există în colecţie. Returnează true dacă s-a modificat colecţia şi false în caz contrar. Clasele care implementează această metodă pot impune condiţii legate de elementele care pot fi adăugate, spre exemplu, elementul *null* poate sau nu să fie adăugat într-o colecţie
boolean addAll(Collection c)	Adaugă în colecţie toate elementele existente în colecţia referită de *c*
void clear()	Şterge toate elementele din colecţie
boolean contains(Object o)	Returnează true dacă colecţia conţine cel puţin un element e astfel încât *(o==null ? e==null : o.equals(e))*
boolean containsAll(Collection c)	Returnează true dacă colecţia conţine toate elementele din colecţia specificată prin *c*
boolean equals(Object o)	Compară colecţia cu obiectul specificat
int hashCode()	Returnează codul hash al colecţiei
boolean isEmpty()	Returnează true dacă nu există nici un element în colecţie
Iterator iterator()	Returnează un iterator care poate fi folosit pentru parcurgerea colecţiei
boolean remove(Object o)	Returnează true dacă s-a şters un element egal cu cel referit de *o* din colecţie. Dacă colecţia conţine elementul *o* duplicat, se va şterge doar un singur element
boolean removeAll(Collection c)	Şterge toate elementele existente în colecţia *c*. Returnează true dacă a avut loc cel puţin o ştergere
boolean retainAll(Collection c)	Reţine în colecţie doar elementele din colecţia *c* (operaţia de intersecţie din teoria mulţimilor). Returnează true dacă colecţia s-a modificat
int size()	Returnează numărul de elemente din colecţie
Object[] toArray()	Returnează un tablou ce conţine toate elementele din colecţie
Object[] toArray(Object[] a)	Returnează un tablou ce conţine toate elementele din colecţie. În acest caz argumentul primit este tipul elementelor din tablou

Tabelul 10.2: Metodele interfeţei Collection.

Prototipul	Descriere
Object get(int index)	Returnează elementul de pe poziţia *index*. Va genera o excepţie *IndexOutOfBoundsException* dacă indexul satisface condiţia *(index < 0 || index >= size())*
Object set(int index, Object element)	Înlocuieşte elementul de pe poziţia *index* cu elementul specificat. Returnează elementul vechi care se afla pe poziţia dată
int lastIndexOf(Object o)	Returnează poziţia ultimei apariţii din listă a elementului referit de *o* sau -1 dacă acesta nu există
Object remove(int index)	Şterge elemetul de pe poziţia specificată din listă

Tabelul 10.3: METODE DIN INTERFAŢA LIST.

Prototipul	Descriere
ArrayList()	Construieşte o listă fără nici un element având capacitatea iniţială de 10 elemente. Dacă se încearcă adăugarea a mai mult de 10 elemente, dimensiunea listei se va modifica automat
ArrayList(Collection c)	Construieşte o listă ce conţine elementele conţinute de colecţia primită ca argument. Capacitatea iniţială este cu 10 procente mai mare decât numărul de elemete din colecţia primită ca argument
ArrayList(int initialCapacity)	Construieşte o listă fără nici un element având capacitatea iniţială de *initialCapacity* elemente

Tabelul 10.4: CONSTRUCTORII CLASEI ARRAYLIST.

Prototipul	Descriere
LinkedList()	Construieşte o listă fără nici un element
LinkedList(Collection c)	Construieşte o listă ce conţine elementele conţinute de colecţia primită ca argument

Tabelul 10.5: CONSTRUCTORII CLASEI LINKEDLIST.

Prototipul	Descriere
void addFirst(Object o)	Adaugă elementul specificat la începutul listei
void addLast(Object o)	Adaugă elementul specificat la sfârşitul listei
Object removeFirst()	Şterge primul element din listă şi returnează o referinţă spre el
Object removeLast()	Şterge ultimul element din listă şi returnează o referinţă spre el
Object getFirst()	Returnează primul element din listă
Object getLast()	Returnează ultimul element din listă

Tabelul 10.6: METODE SPECIFICE CLASEI LINKEDLIST.

ale listei fiind stocate sub forma unei liste înlănţuite. Acest fapt asigură un timp mai bun pentru ştergerea şi inserarea unui element în interiorul listei comparativ cu *ArrayList*. În schimb, accesul aleator la un element din interiorul listei este o operaţie consumatoare de mai mult timp faţă de *ArrayList*.

Ce e mai bine să folosim, *ArrayList* sau *LinkedList*? Răspunsul diferă în funcţie de operaţiile frecvente care se efectuează asupra listelor.

Clasa *LinkedList* are câteva metode în plus faţă de cele din interfaţa *List*, anume metode care permit prelucrarea elementelor aflate la cele două capete ale listei. Aceste metode specifice sunt prezentate în Tabelul 10.6.

Exemplu La începutul lecţiei am spus că trebuie scrisă o aplicaţie care să gestioneze informaţii despre angajaţii unei companii dezvoltatoare de software. În acest scop a fost implementată clasa *Companie* care avea, în prima variantă de implementare, un atribut de tip tablou de angajaţi. Apoi am creat o clasa numită *ListaAngajati* şi am înlocuit în clasa *Companie* tabloul cu un atribut de tip *ListaAngajati*. În exemplul de mai jos, în loc de folosirea unei clase proprii pentru gestionarea angajaţilor am folosit clasa predefinită *ArrayList*.

```
class Companie {

    private String nume;
    private ArrayList angajati;

    public Companie(String nume, int nrAngajati) {
        this.nume = nume;
```

```
     angajati = new ArrayList(nrAngajati);
    }

    public void addAngajat(Angajat a) {
        angajati.add(a);
    }
    ...
}
```

Important Metoda *add* a clasei *ArrayList* are ca parametru o referinţă spre un *Object*. Dar datorită faptului că orice clasă moşteneşte clasa *Object* şi datorită facilităţii oferite de moştenirea de tip, o instanţă a clasei *ArrayList* poate stoca orice tip de obiecte.

Important Se observă că nu există în interfaţa *Collection* metode de genul *public boolean add(tipPrimitiv i)*, unde *tipPrimitiv* desemnează unul dintre tipurile primitive existente.
Datorită mecanismului de autoboxing prezentat în Secţiunea 4.2.2 putem scrie

```
List c = new ArrayList(10);
c.add(6);
```

Dacă dorim să testăm dacă un anumit angajat este sau nu angajat în cadrul companiei, nu trebuie decât să creăm o nouă metodă în clasa *Companie*.

```
public boolean este(Angajat a) {
    return angajati.contains(a);
}
```

Important Polimorfismul este mecanismul datorită căruia metoda *contains* poate testa existenţa unui obiect specificat în interiorul unei colecţii. Concret, pentru colecţia referită de *angajati* metoda *contains* returnează *true* dacă şi numai dacă în colecţie există cel puţin un element *e* astfel încât (a==null ? e==null : a.equals(e)).

Dacă în clasa *Angajat* în loc să suprascriem metoda *equals* din clasa *Object*, am fi creat metoda *public boolean egal(Object o)* cu acelaşi conţinut ca şi metoda *equals* din clasa *Angajat*, nu s-ar fi putut NICIODATĂ testa existenţa unui angajat într-o colecţie predefinită Java – nu s-ar fi testat CORECT egalitatea dintre doi angajaţi din punct de vedere al conţinutului. Atunci când suprascriem metoda *equals* nu facem altceva decât să specificăm ce înseamnă egalitatea dintre două obiecte din punct de vedere al conţinutului – în cazul a două obiecte de tip *Angajat* egalitatea înseamnă nume, respectiv prenume egale.

Dacă dorim să schimbăm departamentul angajatului aflat pe poziţia *pos* în interiorul listei am putea crea în clasa *Companie* metoda de mai jos

```
public void schimbaDepartamentul(int pos, String departamentNou) {
    Angajat a = (Angajat)angajati.get(pos);
    a.schimbaDepartamentul(departamentNou);
}
```

Se observă că accesul la serviciile specifice obiectelor de tip *Angajat* conţinute de listă impune folosirea operatorul *cast*. Însă folosirea operatorului *cast*, pe lângă faptul că este deranjantă, poate introduce erori la rularea programului datorită neconcordanţei dintre tipul real al obiectului existent în colecţie şi tipul spre care facem *cast*. Începând cu varianta 1.5 a limbajului Java s-au introdus tipurile generice care elimină necesitatea conversiilor de tip explicite la lucrul cu colecţii. Detalii despre tipurile generice sunt prezentate în Secţiunea 10.3.

Nu se recomandă ca în clasa *Companie* să existe metode care accesează elemente de pe anumite poziţii ale colecţiei *angajati*. Cauza este simplă: la un moment dat, în interiorul clasei *Companie*, în loc să folosim un obiect *ArrayList* pentru stocarea informaţiilor despre angajaţi, decidem să folosim un obiect instanţă a clasei *HashSet*. Este evident că elementul de pe o anumită poziţie nu ar mai fi posibil de accesat.

10.3 Genericitatea colecţiilor

Despre ce este vorba? În ultima implementare a clasei *Companie* am folosit un obiect *ArrayList* pentru stocarea informaţiilor despre angajaţi. Nu am putut restricţiona tipul elementelor pe care le poate conţine lista referită de atributul *angajati* şi, în consecinţă, următoarea instrucţiune este corectă

```
angajati.add(new Integer(20));
```

Evident că am vrea ca existenţa unor astfel de operaţii să fie semnalată printr-o eroare la compilare. Ei bine, datorită existenţei tipurilor generice putem restricţiona tipul elementelor conţinute de o colecţie.

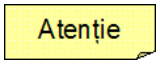
```
class Companie {
    private String nume;
    private ArrayList<Angajat> angajati;
```

```
    public Companie(String nume, int nrAngajati) {
        this.nume = nume;
        angajati = new ArrayList<Angajat>(nrAngajati);
    }
}
```

Declaraţia *private ArrayList<Angajat> angajati* specifică faptul că avem o listă ce poate conţine doar obiecte de tip *Angajat*. Spunem că *ArrayList* este o clasă generică care are un parametru de tip, în acest caz, *Angajat*. De asemenea, parametrul de tip este specificat şi la instanţierea clasei. În acest moment, dacă încercăm să adaugăm în listă elemente de tip *Integer* compilatorul va semnala o eroare.

```
public void schimbaDepartamentul(int pos, String departamentNou) {
    Angajat a = angajati.get(pos);
    a.schimbaDepartamentul(departamentNou);
}
```

Având în vedere că putem stoca în obiectul de tip *ArrayList* doar obiecte de tip *Angajat*, exemplul de mai sus este absolut corect din punct de vedere sintactic iar necesitatea instrucţiunii *cast* a dispărut.



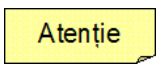
Nu e obligatoriu ca într-un program pe care-l scriem să folosim tipurile generice dar folosirea acestoră măreşte gradul de înţelegere al programului precum şi robusteţea acestuia.

Tipurile generice şi subclasele. Considerăm următoarea secvenţă:

```
ArrayList<Contabil> lContabili = new ArrayList<Contabil>(); //1
ArrayList<Angajat> lAngajati = lContabili; //2
```

Se pune problema corectitudinii linii numerotate cu 2. La prima vedere, linia 2 ar fi corectă. Dar în continuare vom arăta că lucrurile nu sunt chiar aşa.

```
//Daca linia 2 ar fi corecta, am putea adauga intr-o
//lista de contabili orice obiecte de tip Angajat, ceea ce nu am dori
lAngajati.add(new Angajat("Mihai", "Mircea"));
//si undeva s-ar putea atribui unei referinte Contabil un obiect Angajat
Contabil c = lContabili.get(0);
```



Pentru acest exemplu, clasa *Angajat* nu a fost declarată ca fiind abstractă deoarece pentru ilustrarea fenomenului era necesară instanţierea unui obiect de tipul superclasei. Dar ar fi bine ca în programe clase de genul *Angajat* să fie declarate abstracte.

Prototipul	Descriere
boolean hasNext()	Returnează true dacă mai sunt elemente de parcurs în cadrul iteraţiei curente
Object next()	Returnează următorul element din iteraţie
void remove()	Şterge din colecţia care a creat iteratorul ultimul element returnat de iterator

Tabelul 10.7: METODELE INTERFEŢEI ITERATOR.

10.4 Parcurgerea colecţiilor

Spuneam anterior că nu se recomandă ca în interiorul unei clase să existe metode care să acceseze elemente de pe anumite poziţii ale unei colecţii. Dar dacă la un moment dat e nevoie, totuşi, ca un client să aibă acces la elementele colecţiei? Sau dacă în interiorul clasei posesoare a colecţiei se doreşte efectuarea unei anumite operaţii pentru fiecare element existent? Şi în acest caz, schimbarea implementării colecţiei în interiorul clasei va necesita modificări.

Am vazut că în interfaţa *Collection* există metoda *iterator()* ce returnează o referinţă spre un obiect *Iterator*. Iteratorul returnat permite parcurgerea colecţiei într-o ordine bine precizată de fiecare implementare a interfeţei *Collection*.

Metodele interfeţei *Iterator* sunt prezentate în Tabelul 10.7.

Exemple. Parcurgerea elementelor listei de angajaţi în contextul în care nu se folosesc tipurile generice se poate face ca mai jos:

```
Iterator it = angajati.iterator();
while(it.hasNext()) {
    Angajat a = (Angajat)it.next();
    a.schimbaDepartamentul("Contabilitate");
}
```

Dar dacă dorim să scăpăm de instrucţiunea *cast* putem parcurge colecţia în felul următor:

```
Iterator<Angajat> it = angajati.iterator();//1
while(it.hasNext()) { //2
    Angajat a = it.next();
    a.schimbaDepartamentul("Contabilitate");
}
```

Prototipul	Descriere
int compareTo(Object o)	Compară obiectul curent cu cel primit pentru stabilirea unei ordini între cele două obiecte

Tabelul 10.8: METODELE INTERFEŢEI COMPARABLE.

Atenţie Dacă între liniile 1 şi 2 de mai sus, s-ar fi adăugat noi elemente în colecţia referită de *angajati*, ar fi trebuit obţinut un nou iterator.

Instrucţiunea *foreach*. Începând cu Java versiunea 1.5 s-a introdus instrucţiunea denumită foreach cu ajutorul căreia o colecţia de angajaţi se parcurge în felul următor:

```
for(Angajat current: angajati)
    System.out.println(current);
```

Tipărirea elementelor unei colecţii. Putem afişa o colecţie, la fel cum afişăm fiecare obiect, folosind metoda *toString()*. De fapt, în toate implementările interfeţei *Collection*, metoda *toString()* este suprascrisă astfel încât aceasta să returneze reprezentările sub formă de şiruri de caractere a tuturor elementelor conţinute , încadrate între [şi].

10.5 Alte tipuri de colecţii

10.5.1 Mulţimi

Anterior am precizat că toate clasele concrete care au ca supertip interfaţa Set implementează conceptul de mulţime, o mulţime fiind o colecţie ce nu acceptă elemente duplicate. În această secţiune vom exemplifica modul de utilizare al unor clase predefinite pentru lucrul cu mulţimi de obiecte.

În interfaţa *Set* nu există metode în plus faţa de interfaţa *Collection*. Implementarea interfeţei de către clasa *HashSet* nu garantează că elementele vor fi reţinute într-o ordine particulară.

În principiu, pentru operaţii obişnuite cu mulţimi se va folosi clasa *HashSet*. *TreeSet* se utilizează atunci când se doreşte extragerea de elemente într-o anumită ordine. În general, pentru a putea extrage elemente într-o anumită ordine, elementele colecţiei de tip *TreeSet* trebuie să implementeze interfaţa *Comparable*. Clasele predefinite String, clasele înfăşurătoare, chiar şi clasele din suportul pentru lucrul cu colecţii implementează interfaţa *Comparable*. Obiectele ce vor fi adăugate într-o colecţie *TreeSet* trebuie să fie instanţe ale unor clase ce implementează interfaţa *Comparable*.

Exemplu. Fie clasa:

```
class Valoare {

    private int v;

    public Valoare(int v) {
        this.v = v;
    }

    public Valoare(int v) {
        this.v = v;
    }

    public boolean equals(Object o) {
        return (o instanceof Valoare) && ((Valoare)o).v == v;
    }
}
```

După cum se vede, clasa *Valoare* suprascrie metoda *public boolean equals(Object o)*. În acest context, se pune problema afişării efectului produs de codul de mai jos.

```
Set<Valoare> set = new HashSet<Valoare>();
set.add(new Valoare(5));
set.add(new Valoare(5));
System.out.println(set);
```

Am spus că *HashSet* este o clasă ce modelează conceptul de mulţime, o mulţime neputând să conţină elemente duplicate. Atunci ar fi firesc ca tipărirea de mai sus să producă pe ecran textul *[5]* dar, în realitate, dacă rulăm codul de mai sus, vom vedea că se va tipări *[5, 5]*, mulţimea având două elemente "identice".

De fapt, atunci când se testează dacă un element mai e în mulţime contează ca atât metoda *equals* să returneaze egalitate cât şi ca obiectele să aibă acelaşi cod *hash*. Prin urmare soluţia în acest caz este suprascrierea metodei *hashCode*.

```
class Valoare {
    ...
    public int hashCode() {
        return v;
    }
}
```

Atenţie

De obicei, la lucrul cu *HashSet* implicaţia *o1.equals(o2) -> o1.hashCode() == o2.hashCode()* trebuie să fie adevărată.

Prototipul	Descriere
Object put(Object key, Object value)	Asociază obiectul referit de *value* cu cheia specificată de *key*. Dacă în dicţionar mai există stocată cheia *key*, atunci valoarea asociată e înlocuită
Object get(Object key)	Returnează valoarea referită de cheia specificată

Tabelul 10.9: Câteva metode din interfaţa Map.

10.5.2 Dicţionare

Am spus anterior că toate clasele concrete care au ca supertip interfaţa *Map* implementează conceptul de dicţionar cheie-valoare. În această secţiune prezentăm în Tabelul 10.9 câteva metode existente în interfaţa *Map* iar în continuare vom exemplifica modul de folosire al clasei *HashMap*.

Dacă căutarea într-un dicţionar s-ar face liniar, această operaţie ar fi foarte ineficientă din punctul de vedere al timpului necesar efectuării ei. În cadrul dicţionarelor, nu se face o căutare liniară a cheilor ci una bazată pe aşa numita funcţie *hash*. Fiecare cheie are un cod *hash* care e folosit ca index într-un tablou capabil să furnizeze rapid valoarea asociată cheii.

Exemplu. Fie clasa:

```
class Valoare {

    private int v;

    public Valoare(int v) {
        this.v = v;
    }

    public boolean equals(Object o) {
        return (o instanceof Valoare) && ((Valoare)o).v == v;
    }
}
```

În exemplul de mai jos, în dicţionar se vor introduce doi angajaţi, ambii asociaţi cheii cu valoarea *1*.

Nu se recomandă ca într-un dicţionar să existe două chei între care să existe egalitate din punct de vedere al conţinutului.

```
HashMap<Valoare, Angajat> hm = new HashMap<Valoare, Angajat>();
Valoare v1 = new Valoare(1);
Valoare v2 = new Valoare(1);
hm.put(v1, new Contabil("Ionescu","Mircea"));
hm.put(v2, new Contabil("Ion","Mircea"));
System.out.println(hm);
```

Dar ce se va afişa în urma execuţiei?

```
System.out.println(hm.get(new Valoare(1)));
```

Răspunsul este foarte simplu: *null*. Deşi există elemente asociate cheii cu valoarea *1*, cheile neavând acelaşi cod *hash*, elementul asociat indexului corespunzător codului *hash* al cheii din exemplul de mai sus din tabloul în care se păstrează elementele în cadrul implementării interne a clasei *HashMap* este *null*.

Soluţia, şi în acest caz, este suprascrierea metodei *hashCode* pentru clasa Valoare. Atunci, la execuţia codului de mai sus se va afişa *Nume:Ion Departament:contabilitate*. În cartea *Thinking in Java*, Capitolul *Containers in Depth - Overriding hashCode()* este prezentat în detaliu un algoritm pentru generarea de coduri hash corecte!

10.6 Exerciţii rezolvate

Biblioteca

Folosind clasa *ArrayList* creaţi o clasă *Biblioteca* ce poate stoca un număr nelimitat de obiecte de tip *Carte*. O carte are două atribute ce stochează titlul precum şi autorul cărţii iar afişarea acesteia pe ecran va furniza utilizatorului valorile atributelor menţionate.

Clasa *Biblioteca* oferă doar două servicii, unul pentru adăugarea de elemente de tip *Carte* şi altul pentru afişarea elementelor conţinute. Se cere implementarea claselor menţionate precum şi crearea într-o metodă *main* a unei biblioteci ce are trei cărţi. Cărţile ce există în bibliotecă vor fi tipărite.

Rezolvare

```
import java.util.*;

class Carte {
    private String autor, titlu;
```

```
    public Carte(String autor, String titlu) {
        this.autor = autor;
        this.titlu = titlu;
    }

    public String toString() {
        return autor + " " + titlu;
    }
}

class Biblioteca {
    private ArrayList<Carte> carti = new ArrayList<Carte>();

    public void add(Carte c) {
        carti.add(c);
    }

    public String toString() {
        //se poate si parcurge colectia cu iteratori, dar ar fi inutil
        //din moment ce exista metoda toString() din ArrayList
        return carti.toString();
    }

    public static void main(String[] argv) {
        Carte c1 = new Carte("colectiv", "abecedar");
        Carte c2 = new Carte("UPT", "Java");
        Carte c3 = new Carte("INFO", "Java");

        Biblioteca b = new Biblioteca();
        b.add(c1);
        b.add(c2);
        b.add(c3);

        System.out.println(b);
    }
}
```

Fişiere şi Directoare

Respectând cerinţele enunţate şi principiile programării orientate pe obiecte, să se implementeze în Java interfaţa şi clasele descrise mai jos.

Interfaţa *Intrare* conţine:

- o metodă denumită *continut*, fără argumente şi care returnează o referinţă *String*.

Clasa *Fisier* implementează interfaţa *Intrare* şi conţine:

- un atribut de tip *String* denumit *informatie*, specific fiecărui obiect *Fisier* în parte.
- un constructor ce permite setarea atributului anterior cu o valoare *String* dată ca parametru constructorului.
- implementarea metodei *continut* întoarce valoarea atributului *informatie* descris mai sus.

Clasa *Director* implementează interfaţa *Intrare* şi conţine:

- un atribut denumit *intrari* de tip *ArrayList¡Intrare¿*. Câmpul este specific fiecărei instanţe a acestei clase şi se va iniţializa cu un obiect listă gol.
- o metoda *adauga* cu un singur parametru; acesta trebuie să fie declarat în aşa fel încât să poată referi atât obiecte a clasei *Director*, cât şi obiecte a clasei *Fisier* dar să NU poată referi orice fel de obiect Java (spre exemplu, NU va putea referi un obiect *String*). Metoda introduce în lista anterioară referinţa primită ca parametru.
- obligatoriu în implementarea metodei *continut* se parcurge lista intrari şi se apelează metoda *continut* pe fiecare referinţă din lista concatenându-se String-urile întoarse de aceste apeluri; metoda va returna o referinţă spre String-ul rezultat n urma concatenării.

În toată această ultimă clasă, obiectele Fisier si obiectele Director din listă trebuie să fie tratate uniform.

Rezolvare

```
interface Intrare {
    String continut();
}

class Fisier implements Intrare {
    private String informatie;
    public Fisier(String informatie) {
        this.informatie = informatie;
    }
    public String continut() {
        return informatie;
    }
}

class Director implements Intrare {
    private ArrayList<Intrare> intrari = new ArrayList<Intrare>();
```

```
    public void adauga(Intrare intr) {
        intrari.add(intr);
    }
    public String continut() {
        String tmp = "";
        for(Intrare i : intrari) {
            tmp = tmp + i.continut();
        }
        return tmp;
    }
}
```

10.7 Exerciţii

1. Ce credeţi că e mai bine să folosim, *ArrayList* sau *LinkedList*?
2. Scrieţi un program în care se citesc de la tastatură şiruri de caractere până la citirea şirului *STOP*. Şirurile citite se vor stoca într-o colecţie iniţială de tip *LinkedList* ce poate conţine duplicări. Creaţi o nouă colecţie de tip *LinkedList* ce va conţine elementele colecţiei iniţiale, dar fără duplicări. Tipăriţi apoi ambele colecţii.
3. Să se implementeze ierarhia de clase descrisă mai jos:
   - Clasa *Tip*: reprezintă un tip de date abstract
     - Date membru: nu are
     - Metode membru
       - *public String getTip()*: returnează numele clasei sub forma unui şir de caractere precedat de şirul "Tip: "
       - *public String toString()*: afişează valoarea atributului încapsulat de clasele derivate

     Metoda *getTip* nu are iniţial nici o implementare.
   - Clasa Intreg: reprezintă tipul de date întreg (moşteneşte clasa Tip)
     - Date membru: un atribut de tip *int*
     - Metode membru
       - *public String getTip()*
       - *public String toString()*
   - Clasa Sir: reprezintă tipul de date şir de caractere (moşteneşte clasa Tip)
     - Date membru: un atribut de tip *String*
     - Metode membru
       - *public String getTip()*
       - *public String toString()*

- Clasa Colectie: reprezintă tipul de date colecţie de obiecte *Tip*
  - Date membru: un atribut ce stochează elementele colecţiei
  - Metode membru
    - *public String getTip()*
    - *public String toString()*
    - o metodă care testează egalitatea dintre două colecţii din punct de vedere al conţinutului elementelor. Două colecţii sunt considerate a fi egale din punct de vedere al conţinutului dacă ambele conţin acelaşi număr de elemente iar elementele conţinute sunt egale, mai mult, ele sunt stocate în aceeaşi ordine.
    - o metodă pentru adăugarea de elemente în colecţie

  Acest tip de colecţie trebuie implementat astfel încât o colecţie să poată conţine elemente de tip *Colectie*.

  Exemple. Presupunem că avem o colecţie formată din următoarele elemente: 7, 4, Eu, 12. Apelul metodei *toString* trebuie să furnizeze rezultatul (7, 4, Eu, 12).

  Presupunem că avem o colecţie formată din următoarele elemente: 7, 4, Eu, 12 şi colecţia formată din elementele 2 şi 8. Apelul metodei *toString* trebuie să furnizeze rezultatul (7, 4, Eu, 12, (2, 8)). Metoda *toString* din această clasă trebuie să fie implementată urmărind următoarele cerinţe
  - folosirea operatorului instanceof e STRICT interzisă
  - trebuie să existe o variabilă de tip *Iterator* în interiorul metodei

Se va scrie şi o metodă *main* într-o altă clasă în care se va crea o colecţie de obiecte *Tip* ce va avea cel puţin un element de tip *Colectie*, dupa care aceasta se va afişa. Se va testa şi egalitatea elementelor dintre două colecţii.

Bibliografie

1. Gilad Bracha, *Generics in the Java Programming Language*. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.
2. Bruce Eckel. *Thinking in Java, 4th Edition*. Prentice-Hall, 2006. Capitolul Containers in Depth.
3. Sun Microsystems Inc., *Online Java 1.5 Documentation, http://java.sun.com/j2se/1.5.0/docs/api/*, 2005.