Colectii

De la POO

Salt la: Navigare, căutare

Cuprins

[ascunde] [ascunde]

Collections Framework

In pachetul **`java.util`** (pachet standard din JRE) exista o serie de clase pe care le veti gasi folositoare.

Collections Framework este o arhitectura unificata pentru reprezentarea si manipularea colectiilor. Ea contine:

- **interfete**: permit colectiilor sa fie folosite independent de implementarile lor
- **implementari**
- **algoritmi**: metode de prelucrare (cautare, sortare) pe colectii de obiecte oarecare. Algoritmii sunt *polimorfici*: un astfel de algoritm poate fi folosit pe implementari **diferite** de colectii, deoarece le abordeaza la nivel de interfata.

Colectiile ofera implementari pentru urmatoarele tipuri:

- **multime** (ordinea elementelor este neimportanta)
- **lista** (ordinea elementelor conteaza)
- **tabel asociativ** (perechi cheie-valoare)

si pot lucra cu **orice** tip de obiecte (implementarile sunt *generice*).

Exista o interfata, numita **`Collection`**, pe care o implementeaza majoritatea claselor ce desemneaza colectii din `java.util`. Aceasta contine metode utile precum:

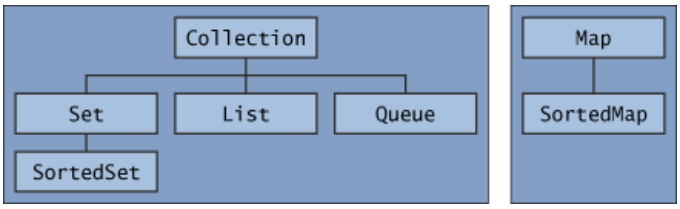
- `add(T)` - adauga un obiect
- `addAll(Collection)` - adauga o intreaga alta colectie
- `clear()` - sterge toate elementele
- `contains(T)` - verifica daca obiectul respectiv exista. Se foloseste metoda `equals`!!
- `iterator()` - intoarce un **`Iterator`** cu care poate fi parcursa colectia
- `remove(T)` - sterge elementul din colectie
- `size()` - intoarce numarul de elemente din colectie
- `toArray()` - intoarce un vector de obiecte: `Object[]`
- `isEmpty` etc.

Explicatii suplimentare gasiti pe Java Tutorials - `Collection`.

Exemplul de mai jos construieste o lista populata cu nume de studenti:

```
Collection names = new ArrayList();
names.add("Andrei");
names.add("Matei");
```

Mai jos este prezentata o imagine de ansamblu asupra tipurilor standard de colectii Java. Nodurile reprezinta interfetele ce vor fi discutate pe parcursul laboratorului.



Parcurgerea colectiilor

Colectiile pot fi parcurse (element cu element) folosind:

- **iteratori**
- o constructie `for` speciala (cunoscuta sub numele de **for-each**)

Iteratori

Un iterator este un obiect care permite traversarea unei colectii si modificarea acesteia (ex: stergere de elemente) in mod selectiv. Puteti obtine un iterator pentru o colectie, apeland metoda sa `iterator()`. Interfata **`Iterator`** este urmatoarea:

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();    // optional
}
```

Metodele au urmatorul comportament:

- `hasNext` intoarce `true` daca mai **exista** elemente neparcurse inca de iteratorul respectiv
- `next` intoarce **urmatorul** element
- `remove` elimina din colectie ultimul element intors de `next`. In mod evident, `remove` nu poate fi apelat decat o singura data dupa un apel `next`. Daca aceasta regula nu este respectata, vom primi o eroare.

Ne putem imagina ca un iterator se pozitioneaza **intre** elementele colectiei. Initial, cursorul sau **precede** primul element, astfel ca primul apel `next` va intoarce primul element.

Atentie: Metoda `remove` este singura modalitate **SIGURA** de a **inlatura** un element dintr-o colectie in timpul **parcurgerii** acesteia. Orice alta metoda are un comportament neprecizat (nu putem garanta ca stergerea va avea loc, sau ca elementul sters va fi cel pe care chiar doream sa-l stergem).

Este util sa folosim **iteratori** cand dorim:

- **stergerea** elementului curent, in timpul iterarii
- Cand dorim sa iteram mai **multe** colectii in paralel.

Exemplu de folosire a unui iterator:

```
Collection c  = new ArrayList();
Iterator   it = c.iterator();

while (it.hasNext()) {
     //verificari asupra elementului curent: it.next();
     it.remove();
}
```

Acest cod este *polimorfic*, ceea ce inseamna ca functioneaza pentru **orice** colectie, indiferent de implementare.

For-each

Aceasta constructie permite (intr-o maniera expeditiva) traversarea unei colectii. **for-each** este foarte similar cu **`for`**. Urmatorul exemplu parcurge elementele unei colectii si le afiseaza.

```
Collection collection = new ArrayList();
for (Object o : collection)
```

```
    System.out.println(o);
```

Constructia **for-each** se bazeaza, in spate, pe un iterator, pe care il ascunde. Prin urmare **nu** putem sterge elemente in timpul iterarii.

In aceasta maniera pot fi parcursi si **vectori** oarecare. De exemplu, `collection` ar fi putut fi definit ca `Object[]`.

Genericitate

Fie urmatoarea portiune de cod:

```
Collection c = new ArrayList();
c.add("Test");

Iterator it  = c.iterator();

while (it.hasNext()) {
    String s = it.next();         // EROARE: next intoarce Object si este nevoie de cast explicit la String
    String s = (String)it.next(); // OK
}
```

Am definit o colectie `c`, de tipul `ArrayList` (pe care il vom examina intr-o sectiune urmatoare). Apoi, am adaugat in colectie un element de tipul `String`. Am realizat o parcurgere folosind un iterator, si am incercat obtinerea elementului nostru folosind apelul: `String s = it.next();`. Functia `next` insa intoarce un obiect de tip `Object`. Prin urmare apelul va esua. Varianta corecta este `String s = (String)it.next();`.

Conversia explicita (din exemplul de mai sus), ridica urmatoarele probleme:

- este **neplacut** sa realizam un *cast* pentru fiecare acces la un element. De obicei, programatorii stiu ce tipuri folosesc in listele lor.
- cu toate acestea, programatorii pot face **greseli**; spre exemplu, linia din codul de mai sus ar putea fi inlocuita cu urmatoarea: `Integer i = (Integer)it.next();` . Cum `Integer` mosteneste `Object`, conversia este acceptata de compilator. Vom obtine insa o **eroare** la **runtime**, intrucat obiectul intors de iterator nu este un intreg.
- desi intentionam sa retinem *string*-uri in colectia noastra, nu putem impiedica adaugarea unui element de **alt** tip, ca `Integer`, caz in care cast-ul "corect" de mai sus va esua.

Am dori asadar sa inlaturam aceasta ambiguitate, si sa specificam, din start, ce tipuri de date dorim sa pastram intr-o colectie. Acest lucru este posibil. Exemplu:

```
Collection<String> c = new ArrayList<String>();
c.add("Test");
c.add(2);       // EROARE!
Iterator<String> it = c.iterator();

while (it.hasNext()) {
     String s = it.next();
}
```

Observati modul de definire a colectiei. Constructia `Collection<String>` *spune* ca nu dorim sa creem o colectie oarecare ci una care sa contina tipuri `String`. Spunem ca interfata `Collection` este o interfata **generica**, ce primeste un parametru de tip (in exemplul nostru, `String`).

Am putea fi tentati sa credem ca genericitatea nu ajuta foarte mult. Aparent, am schimbat un cast explicit printr-o parametrizare. Lucrurile insa nu stau chiar asa. Folosind genericitate, **corectitudinea** tipurilor poate fi **verificata** la compilare. Un apel de genul `Integer i = (Integer)it.next();` va produce o eroare la compilare. Compilatorul *stie* acum ca iteratorul parcurge elemente de tip `String` si, prin urmare, `it.next()` va intoarce un `String`. De asemenea, adaugarea unui element de **alt** tip va fi interzisa.

In cele ce urmeaza vom folosi colectii parametrizate. Acest lucru este un *good-practice* si este incurajat calduros.

Interfata List

O lista este o colectie **ordonata**. Listele pot contine elemente **duplicate**. Pe langa operatiile mostenite de la `Collection`, interfata **`List`** defineste urmatoarele operatii:

- `T get(int index)` - intoarce elementul de la pozitia `index`
- `T set(int index, T element)` - modifica elementul de la pozitia `index`
- `void add(int index, T element)` - adauga un element la pozitia `index`
- `T remove(int index)` - sterge elementul de la pozitia `index`

Alaturi de `List`, este definita interfata **`ListIterator`**, ce extinde interfata `Iterator` cu metode de parcurgere in ordine inversa.

Explicatii suplimentare gasiti pe Java Tutorials - `List`.

`List` poseda doua **implementari** standard:

- `ArrayList` - implementare sub forma de vector. Accesul la elemente se face in timp constant: `O(1)`
- `LinkedList` - implementare sub forma de lista dublu inlantuita. Prin urmare, accesul la un element nu se face in timp constant, fiind necesara o parcurgere a listei: `O(n)`.

Algoritmi implementati:

- `sort` - realizeaza sortarea unei liste
- `binarySearch` - realizaeaza o cautare binare a unei valori intr-o lista

In general, algoritmii pe colectii sunt implementati ca metode statice in clasa **`Collections`**.

Atentie: Nu confundati interfata `Collection` cu clasa `Collections`. Spre deosebire de prima, a doua este o clasa ce contine exclusiv metode statice. Aici sunt implementate diverse operatii asupra colectiilor.

Iata un exemplu de folosire a sortarii:

```
List<Integer> l = new ArrayList<Integer>();
l.add(5);
l.add(7);
l.add(9);
l.add(2);
l.add(4);

Collections.sort(l);
System.out.println(l);
```

Mai multe detalii despre algoritmi pe colectii gasiti pe Java Tutorials - Algoritmi pe liste.

Compararea elementelor

Rularea exemplului de sortare ilustrat mai sus arata ca elementele din `ArrayList` se sorteaza crescator. Ce se intampla cand dorim sa realizam o sortare particulara pentru un tip de date complex? Spre exemplu, dorim sa sortam o lista `ArrayList<Student>` dupa media anilor. Sa presupunem ca `Student` este o clasa ce contine printre membrii sai o variabila ce retine media anilor.

Acest lucru poate fi realizat folosind interfetele:

- `Comparable`
- `Comparator`

Comparable

Pentru a putea compara direct o instanta a unui tip (clasa definita de noi) cu o alta, este necesar ca tipul sa implementeze interfata **`Comparable`**. Ea contine o singura metoda:

```
int compareTo(Object o)
```

`compareTo` intoarce:

- `>0`, daca instanta curenta este *mai mare* decat cea primita ca parametru
- `0`, daca ele sunt *egale*
- `<0`, daca instanta curenta este *mai mica* decat cea primita ca parametru

Intuitiv, faptul ca o clasa `Student` (spre exemplu) implementeaza `Comparable`, inseamna ca tipul `Student` defineste o relatie de ordine peste instantele sale. Exemplu:

```
import java.util.*;

class Student implements Comparable<Student> {
    double avg;

    public Student (double avg) {
        this.avg = avg;
    }

    @Override
    public int compareTo(Student s) {
        if (avg > s.avg )
```

```
            return 1;

        if (avg == s.avg )
            return 0;

        return -1;
    }

    @Override
    public String toString() {
        return "[" + avg + "]";
    }
}

public class Test {
    public static void main(String[] args) {
        List<Student> l = new ArrayList<Student>();
        l.add(new Student(4.5));
        l.add(new Student(8.7));
        l.add(new Student(5.9));
        l.add(new Student(9.5));
        l.add(new Student(3.0));

        Collections.sort(l);
        System.out.println(l);
    }
}
```

Observati constructia `@Override` ce preceda functia `compareTo`. Aceasta specifica faptul ca metoda `compareTo` **supradefineste** o metoda dintr-o clasa mostenita sau interfata implementata.

Atentie: `Object` defineste o metoda `equals(Object o)`. Este important ca `equals` si `compareTo` sa fie **consistente** una in raport cu cealalta. Daca `(!a.equals(b) && a.compareTo(b) == 0)`, putem avea probleme in implementarile structurilor noastre de date (spre exemplu `SortedSet`).

Comparator

Spre deosebire de `Comparable` care, implementata de o clasa, marca faptul ca instantele sale sunt comparabile, **`Comparator`** desemneaza o entitate externa care realizeaza o comparatie intre doua obiecte oarecare.

Metoda de interes definita in interfata `Comparator` este:

- `int compare(Object o1, Object o2)`: valoarea intoarsa respecta conventiile metodei `compareTo` a interfetei `Comparable`

Din nou, este important de retinut ca `compare` si `equals` trebuie sa fie **reciproc-consistente**.

Intuitiv, faptul ca o clasa implementeaza `Comparator` inseamna ca ea se comporta ca un comparator pentru obiecte de un anumit tip. Exemplu:

```
import java.util.*;

class Student {
    double avg;

    public Student(double avg) {
        this.avg = avg;
    }

    @Override
    public String toString() {
        return "[" + avg + "]";
    }
}

class StudentComparator implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        return (int)(s1.avg - s2.avg);
    }
}

public class Test {
    public static void main(String[] args) {
        List<Student> l = new ArrayList<Student>();
        l.add(new Student(4.5));
        l.add(new Student(8.7));
        l.add(new Student(5.9));
        l.add(new Student(9.5));
        l.add(new Student(4.9));
```

```
        Collections.sort(l, new StudentComparator());
        System.out.println(l);
    }
}
```

Observati faptul ca functia statica `sort` este supraincarcata; ea poate primi un obiect de tip `Comparator`.

Explicatii suplimentare gasiti pe Java Tutorials - Object Ordering.

Interfata Set

Un **Set** (multime) este o colectie ce nu poate contine elemente duplicate. Interfata `Set` contine doar metodele mostenite din `Collection`, la care adauga restrictii astfel incat elementele duplicate sa nu fie poata fi adaugate.

Avem trei implementari utile pentru Set:

- **HashSet**: memoreaza elementele sale intr-o **tabela de dispersie** (*hash table*); este implementarea cea mai performanta, insa **nu** avem garantii asupra **ordinii** de parcurgere. Doi iteratori **diferiti** pot parcurge elementele multimii in ordine **diferita**.
- **TreeSet**: memoreaza elementele sale sub forma de **arbore rosu-negru**; elementele sunt **ordonate** pe baza valorilor sale. Implementarea este mai **lenta** decat `HashSet`.
- **LinkedHashSet**: este implementat ca o tabela de dispersie. Diferenta fata de `HashSet` este ca `LinkedHashSet` mentine o lista dublu-inlantuita peste toate elementele sale. Prin urmare (si spre deosebire de `HashSet`), elementele raman in **ordinea** in care au fost inserate. O parcurgere a `LinkedHashSet` va gasi elementele mereu in **aceasta** ordine.

Atentie: Implementarea `HashSet`, care se bazeaza pe o **tabela de dispersie**, calculeaza codul de dispersie al elementelor pe baza metodei `hashCode`, definita in clasa `Object`. De aceea, doua obiecte **egale**, conform functiei `equals`, trebuie sa intoarca **acelasi** rezultat din `hashCode`.

Explicatii suplimentare gasiti pe Java Tutorials - Set.

Interfata Map

Un **Map** este un obiect care mapeaza **chei** pe **valori**. Intr-o astfel de structura **nu** pot exista chei duplicate. Fiecare cheie este mapata la exact o valoare. `Map` reprezinta o modelare a conceptului de functie: *primeste* o entitate ca parametru (cheia), si intoarce o alta entitate (valoarea).

Cele trei implementari pentru `Map` sunt:

- **HashMap**
- **TreeMap**
- **LinkedHashMap**

Particularitatile de implementare corespund celor de la `Set`.

Exemplu de folosire:

```
class Student {
    String name; // numele studentului
    float avg;   // media

    public Student(String name, float avg) {
        this.name = name;
        this.avg  = avg;
    }

    public String toString() {
        return "[" + name + ", " + avg + "]";
    }
}

public class Test {
    public static void main(String[] args) {

        Map<String,Student> students = new HashMap<String,Student>();

        students.put("Matei",  new Student("Matei",  4.90F));
        students.put("Andrei", new Student("Andrei", 6.80F));
        students.put("Mihai",  new Student("Mihai",  9.90F));

        System.out.println(students.get("Mihai")); // elementul avand cheia "Andrei"

        //adaugam un element cu o cheie existenta
```

```
        System.out.println(students.put("Andrei", new Student("", 0.0F)));
        //put intoarce vechiul element,

        //si apoi il suprascrie
        System.out.println(students.get("Andrei"));

        //remove intoarce elementul sters
        System.out.println(students.remove("Matei"));

        //afisare a structurii
        System.out.println(students);
    }
}
```

Interfata **`Map.Entry`** desemneaza o pereche (cheie, valoare) din map. Metodele caracteristice sunt:

- `getKey`: intoarce cheia
- `getValue`: intoarce valoarea
- `setValue`: permite stabilirea valorii asociata cu aceasta cheie

O **iterare** obisnuita pe un map se va face in felul urmator:

```
for (Map.Entry<String, Student> entry : students.entrySet())
    System.out.println("Media studentului " + entry.getKey() + " este " + entry.getValue().getAverage());
```

In bucla *for-each* de mai sus se ascunde, de fapt, iteratorul multimii de perechi, intoarse de `entrySet`.

Explicatii suplimentare gasiti pe Java Tutorials - Map.

Alte interfete

Queue

`Queue` defineste operatii specifice pentru **cozi**:

- insertia unui element
- stergerea unui element
- operatii de *inspectie* a cozii

Implementari utilizate frecvente pentru `Queue`:

- `LinkedList`: pe langa `List`, `LinkedList` implementeaza si `Queue`
- `PriorityQueue`;

Explicatii suplimentare gasiti pe Java Tutorials - Queue

SortedMap si SortedSet

Ambele sunt interfete care, in plus fata de parintii lor `Map`, respectiv `Set`, isi mentin elementele **sortate**. Orice operatie de adaugare sau stergere va **conserva** aceasta proprietate.

- Un `SortedMap` isi mentine elementele ordonate dupa chei, folosind compararea naturala a cheilor sau un `Comparator` trimis ca parametru la crearea `SortedMap`-ului. O parcurgere a `SortedMap` va intoarce mereu elementele in ordine. Exemplu: `TreeMap`
- Aceleasi observatii se aplica si pentru `SortedSet`. Exemplu: `TreeSet`

Explicatii suplimentare gasiti pe Java Tutorials - SortedSet, SortedMap

Conversia intre vectori si colectii

Sa ne imaginam situatia in care avem la dispozitie un vector si dorim sa il pasam unei metode care asteapta o colectie sau viceversa. Similar, sa presupunem ca dorim sa construim manual o colectie, element cu element. In acest caz secund, utilizarea unui vector este mai potrivita, deoarece acestia pot fi definiti prin enumerarea elementelor, ca in exemplul de mai jos:

```
// Populatea manuala a unei colectii - prea multa vorbarie :)
Collection<String> namesList = new ArrayList<String>();
names.add("John");
names.add("Marry");
...
```

```
// Populatea manuala a unui vector - mult mai compact.
String[] namesArray = {"John", "Mary", ...};
```

Dar, ne lovim de aceeasi problema: in final, trebuie sa obtinem o colectie. Din fericire, biblioteca Java permite realizarea conversiilor in ambele sensuri. Astfel, pentru realizarea unei conversii **de la o colectie la un vector**, putem intrebuinta metoda `Collection.toArray(T[])`, ca in exemplul urmator:

```
// toArray primeste ca parametru un vector in care incearca sa depuna elementele colectiei, daca exista suficient spatiu.
// Altfel, alt vector mai incapator este alocat, avand acelasi tip cu cel al vectorului dat ca parametru.
String[] namesArray = namesList.toArray(new String[0]);
```

Invers, avem la dispozitie metoda `Arrays.asList(T...)`. Iata un exemplu:

```
List<String> namesList = Arrays.asList(namesArray);
```

Exercitii

1. (**1p**) Instantiati o colectie care sa **nu** permita introducerea elementelor duplicate, folosind o implementare corespunzatoare din biblioteca. La introducerea unui element existent, semnalati eroare. Colectia va retine `String`-uri si va fi **parametrizata**.
2. (**2p**) Creati o clasa `Student`.
   - Adaugati urmatorii membri:
     - **campurile** `nume` (de tip `String`) si `medie` (de tip `float`)
     - un **constructor** care ii initializeaza
     - metoda `toString`.
   - Modificati exercitiul anterior astfel incat colectia aleasa de voi sa retina obiecte de tip `Student`. Adaugati elemente duplicate, instantiindu-le, de fiecare data, cu `new`. Ce observati?
   - Prelucrati implementarea de mai sus astfel incat colectia sa nu permita duplicate, dupa un criteriu ales de voi.
     - Supradefiniti metoda `equals` a clasei `Student` si incercati din nou. De ce situatia ramane neschimbata?
     - Supradefiniti metoda `hashCode` si incercati din nou.

     Hint: `Set.add`, `Object.equals`, `Object.hashCode`
3. (**2p**) Plecand de la implementarea exercitiului anterior, realizati urmatoarele modificari:
   - Supraincarcati, in clasa `Student`, metoda `equals`, cu o varianta care primeste un parametru `Student`, si care intoarce, intotdeauna, `false`
   - Iterati pe colectia de la exercitiul 3 si afisati, la fiecare pas, `element.equals(element)` si `((Object)element).equals(element)`. Cum explicati comportamentul observat? Iteratorul va fi, si el, **parametrizat**.
4. (**3p**) Scrieti o clasa, ce va reprezenta un `Map` pentru retinerea studentilor dupa medie.
   - `Map`-ul va contine chei de la 0 la 10 (corespunzatoare mediilor posibile).
   - Asociati fiecarei chei o lista (`List`) care va retine toti studentii cu media rotunjita egala cu cheia. Consideram ca un student are media rotunjita 8 daca media sa este in intervalul [7.50, 8.49].
   - `Map`-ul vostru va mentine cheile (mediile) **ordonate descrescator**. Extindeti o implementare potrivita a interfetei `Map`, care sa permita acest lucru, si folositi un `Comparator` pentru stabilirea ordinii cheilor.
   - Definiti in clasa metoda `add(Student)`, ce va adauga un student in lista corespunzatoare mediei lui. Daca, in prealabil, nu mai exista niciun student cu media respectiva (rotunjita), atunci lista va fi creata **la cerere**.
   - Populati-l cu cativa studenti.
   - Iterati pe map, folosind varianta specifica de **for-each**, si sortati **alfabetic** fiecare lista de studenti. In prealabil, clasa `Student` va implementa interfata `Comparable` si va da o implementare corespunzatoare a metodei `compareTo`.
5. (**2p**) Creati o clasa care mosteneste `HashSet<Integer>`.
   - Definiti in aceasta clasa o variabila membru care retine numarul total de elemente adaugate. Pentru a contoriza acest lucru, supradefiniti metodele `add` si `addAll`. Pentru adaugarea efectiva a elementelor, folositi implementarile din clasa parinte (`HashSet`).
   - Testati, folosind atat `add` cat si `addAll`. Ce observati? Corectati daca este cazul.
   - Modificati implementarea astfel incat clasa voastra sa mosteneasca `LinkedList<Integer>`. Ce observati? Ce **concluzii** trageti?

     Hint: `Collection.add`, `Collection.addAll`
6. (**2p**) Realizati un test comparativ de stress intre `ArrayList` si `LinkedList`.
   - Adaugati un numar foarte mare de elemente (aleatoare) (de ordinul a 10.000 - 100.000).
   - Realizati un numar foarte mare de adaugari/stergeri in pozitii aleatoare. Inregistrati timpul total de executie, atat pentru `ArrayList` cat si pentru `LinkedList`.
   - Separat, realizati un numar foarte mare de accese (`get`). Ce observati? Concluzii.

- **Solutii**

Referinte

- Colectii pe Java Tutorials

Adus de la "http://cursuri.cs.pub.ro/~poo/wiki/index.php/Colectii"