Curs 14
Colectii

Ce sunt colectiile ?

Definitie

O *colectie* este un obiect care grupeaza mai multe elemente într-o singura unitate. Prin colectii vom avea acces la tipuri de date cum ar fi vectori, multimi, tabele de dispersie, etc. Colectiile sunt folosite pentru memorarea si manipularea datelor, precum si pentru transmiterea datelor de la o metoda la alta.

In Java colectiile sunt tratate intr-o maniera unitara, fiind organizate intr-o arhitectura ce cuprinde:

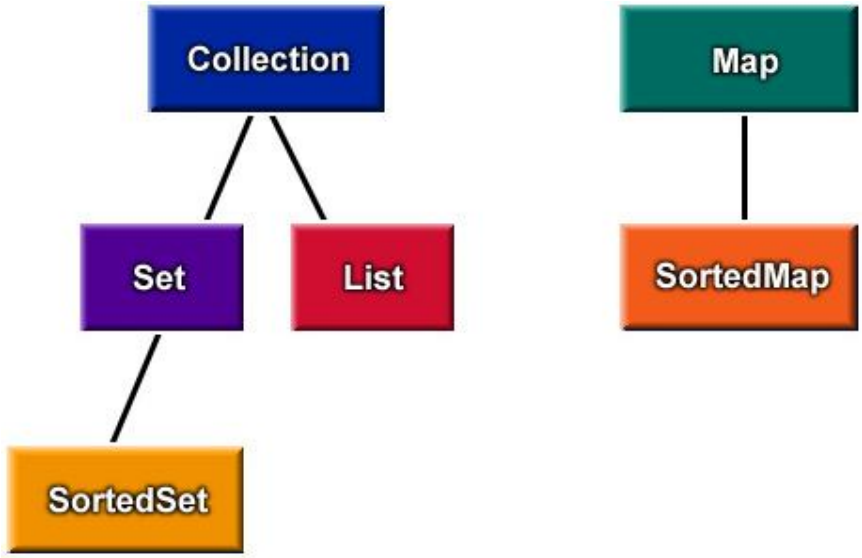
- **Interfete**: tipuri abstracte de date ce descriu colectiile. Interfetele permit utilizarea colectiilor independent de detaliile implementarilor.
- **Implementari**: implementari concrete ale interfetelor ce descriu colectii. Aceste clase reprezinta *tipuri de date reutilizabile*.
- **Algoritmi**: metode care efectueaza diverse operatii utile cum ar fi cautarea, sortarea definite pentru obiecte ce implementeaza interfete ce descriu colectii. Acesti algoritmi se numesc si *polimorfici* deoarece aceeasi metoda poate fi folosita pe implementari diferite ale unei colectii. Aceste tipuri de algoritmii descriu notiunea de *functionalitate reutilizabila*.

Dintre avantajele oferite de utilizarea colectiilor amintim:

- **Reducerea efortului de programare**: prin punerea la dispozitia programatorului a unui set de tipuri de date si algoritmi ce modeleaza structuri si operatii des folosite in aplicatii.
- **Cresterea vitezei si calitatii programului**: implementarile efective ale colectiilor sunt de inalta performanta si folosesc algoritmi cu timp de lucru optim. In scrierea unei aplicatii putem sa ne concentram eforturile asupra problemei in sine si nu asupra modului de reprezentare si manipulare a informatiilor.

Interfete ce descriu colectii

Curs 14
Interfetele ce descriu colectii reprezinta nucleul mecanismului de lucru cu colectii. Scopul acestor interfete este de a permite utilizarea colectiilor independent de modul lor de implementare. Ierarhia lor este prezentata in imaginea de mai jos:



Mai jos sunt descrise structurile de date modelate de aceste interfete:

Collection

Collection descrie un grup de obiecte numite si *elementele* sale. Unele implementari ale acestei interfete permit existenta elementelor duplicate, alte implementari nu. Unele au elementele ordonate, altele nu. Modeleaza o colectie la nivelul cel mai general. In JDK nu exista nici o implementare directa a acestei interfete, ci exista doar implementari ale unor subinterfete mai concrete cum ar fi `Set` sau `List`.

Set

Modeleaza notiunea de *multime* în sens matematic. O multime nu poate avea elemente duplicate.

List

Descrie *liste (secvente)* de elemente indexate. Listele pot contine duplicate si permit un control precis asupra pozitiei unui element prin intermediul indexului acelui element.
O clasa independenta ce implementeaza o functionalitate asemanatoare este clasa `Vector`.

Map

Implementarile acestei interfete sunt obiecte ce asociaza fiecarui element o cheie unica. Nu pot contine asadar chei duplicate si fiecare chei este asociata la un singur element.
O clasa independenta ce implementeaza o functionalitate asemanatoare este clasa `HashTable`.

SortedSet

Este asemanatoare cu interfata `Set` la care se adauga faptul ca elementele dintr-o astfel de colectie sunt ordonate ascendent. Pune la dispozitie operatii care beneficiaza de avantajul ordonarii elementelor.

SortedMap

Este asemanatoare cu interfata `Map` la care se adauga faptul ca multimea cheilor dintr-o astfel de colectie este ordonata ascendent.

Interfata Collection

Interfata `Collection` modeleaza un grup de obiecte numite *elemente*. Scopul acestei interfete este de a folosi colectii la un nivel de maxima generalitate. In definitia interfetei vom observa ca metodele se impart in trei categorii.

```
public interface Collection {
      // Operatii de baza la nivel de element
      int size();
      boolean isEmpty();
      boolean contains(Object element);
      boolean add(Object element);    // Optional
      boolean remove(Object element); // Optional
      Iterator iterator();

      // Operatii la nivel de colectie
      boolean containsAll(Collection c);
      boolean addAll(Collection c);    // Optional
      boolean removeAll(Collection c); // Optional
      boolean retainAll(Collection c); // Optional
      void clear();                    // Optional

      // Operatii de conversie in vector
      Object[] toArray();
      Object[] toArray(Object a[]);
}
```

Interfata Set

Modeleaza notiunea de *multime* în sens matematic. O multime nu poate avea elemente duplicate. Defineste aceleasi metode ca interfata `Collection`. Doua dintre clasele care ofera implementari concrete ale acestei interfete sunt **`HashSet`** si **`TreeSet`**. (vezi "Implementari")

Interfata List

Interfata `List` descrie *liste (secvente)* de elemente indexate. Listele pot contine duplicate si permit un control precis asupra pozitiei unui element prin intermediul indexului acelui element. In plus fata de elementele definite de interfata `Collection` avem metode pentru:

- acces pozitional
- cautare (aflarea indexului unui element)
- iterare ordonata
- extragerea unei subliste

Definitia interfetei este data mai jos:

```
public interface List extends Collection {
      // Acces pozitional
      Object get(int index);
```

```
        Object set(int index, Object element);           // Optional
        void add(int index, Object element);             // Optional
        Object remove(int index);                        // Optional
        abstract boolean addAll(int index, Collection c); // Optional

        // Cautare
        int indexOf(Object o);
        int lastIndexOf(Object o);

        // Iterare
        ListIterator listIterator();
        ListIterator listIterator(int index);

        // Extragere sublista
        List subList(int from, int to);
    }
```

Doua clase care implementeaza interfata `List` sunt **`ArrayList`** si **`Vector`**.

Interfata Map

Implementarile acestei interfete sunt obiecte ce asociaza fiecarui element o cheie unica. Nu pot contine asadar chei duplicate si fiecare chei este asociata la un singur element.
Definitia interfetei este data mai jos:

```
    public interface Map {
        // Operatii de baza la nivel de element
        Object put(Object key, Object value);
        Object get(Object key);
        Object remove(Object key);
        boolean containsKey(Object key);
        boolean containsValue(Object value);
        int size();
        boolean isEmpty();

        // Operatii la nivel de colectie
        void putAll(Map t);
        void clear();

        // Vizualizari ale colectiei
        public Set keySet();
        public Collection values();
        public Set entrySet();

        // Interfata pentru manipularea unei inregistrari
        public interface Entry {
            Object getKey();
            Object getValue();
            Object setValue(Object value);
        }
    }
```

Clase care implementeaza interfata `Map` sunt **`HashMap`**, **`TreeMap`** si **`Hashtable`**.

Interfata SortedSet

Este asemanatoare cu interfata `Set` la care se adauga faptul ca elementele dintr-o astfel de colectie sunt ordonate ascendent conform ordinii lor naturale, sau conform cu ordinea data de un comparator specificat la crearea colectiei.
Este subclasa a interfetei `Set`, oferind metode suplimentare pentru:

- extragere de subliste

- aflarea primului/ultimului element din lista
- aflarea comparatorului folosit pentru ordonare

Definitia interfetei este data mai jos:

```
public interface SortedSet extends Set {
    // Subliste
    SortedSet subSet(Object fromElement, Object toElement);
    SortedSet headSet(Object toElement);
    SortedSet tailSet(Object fromElement);

    // Capete
    Object first();
    Object last();

    Comparator comparator();
}
```

Interfata SortedMap

Este asemanatoare cu interfata `Map` la care se adauga faptul ca multimea cheilor dintr-o astfel de colectie este ordonata ascendent conform ordinii naturale, sau conform cu ordinea data de un comparator specificat la crearea colectiei.
Este subclasa a interfetei `Map`, oferind metode suplimentare pentru:

- extragere de subtabele
- aflarea primei/ultimei chei
- aflarea comparatorului folosit pentru ordonare

Definitia interfetei este data mai jos:

```
public interface SortedMap extends Map {

    SortedMap subMap(Object fromKey, Object toKey);
    SortedMap headMap(Object toKey);
    SortedMap tailMap(Object fromKey);

    Object first();
    Object last();

    Comparator comparator();
}
```

Implementari ale colectiilor

Clasele de baza care implementeaza interfete ce descriu colectii sunt prezentate in tabelul de mai jos. Numele lor este de forma <`Implementare`><`Interfata`>, unde 'implementare' se refera la structura de date folosita.

		Implementari			
Interfete	**Set**	HashSet		TreeSet	

	List		ArrayList		LinkedList
	Map	HashMap		TreeMap	

JDK 1.2 furnizeaza câte doua clase ce implementeaza fiecare tip de colectie, în fiecare caz prima implementare fiind cea de baza, care va fi in general folosita. Acestea sunt: `HashSet, ArrayList` si `HashMap`.

Clasele care descriu colectii au multe trasaturi comune cum ar fi:

- permit elementul `null`
- sunt serializabile
- au definita metoda `clone`
- au definita metoda `toString`, care returneaza o reprezentare ca sir de caractere a colectiei respective
- permit crearea iteratorilor pentru parcurgere
- implementarea interfetelor este indirecta, în sensul ca au o

Implementarea interfetelor este indirecta în sensul ca aceste clase au superclase abstracte care ofera implementari concrete pentru majoritatea metodelor definite de interfete. Cele mai importante superclase sunt **`AbstractCollection`** si **`AbstractMap`**, din care sunt apoi extinse clasele abstracte **`AbstractList`** si **`AbstractSet`**,respectiv **`AbstractMap`**.
Clasele prezentate in tabelul de mai sus sunt extensii concrete ale claselor abstracte aminitite.
Singura interfata care nu are nici o implementare este `Collection`.

Folosirea eficienta a colectiilor

Dupa cum am vazut, fiecare interfata ce descrie o colectie are câte doua implementari, dintre care una este de baza, fiind folosita in 90% din cazuri. De exemplu, interfata `List` este implementata de clasele `ArrayList` si `LinkedList`, prima fiind cea mai folosita. De ce exista atunci si clasa `LinkedList` ?
Raspunsul consta in faptul ca implementari diferite ale interfetelor pot oferi performante mai bune in functie de situatie, prin realizarea unor compromisuri între spatiul necesar pentru reprezentarea datelor, rapiditatea regasirii acestora si timpul necesar actualizarii colectiei in cazul unor modificari.
Sa consideram urmatoarele exemple ce creeaza o lista folosind `ArrayList`, respectiv `LinkedList` si executa diverse operatii pe ea.

```
//exemplul 1
import java.util.*;
public class List1 {
        public static void main(String(args[]) {
                List lst = new ArrayList();
                //List lst = new LinkedList();
                final int N = 25000;
                for(int i=0; i < N; i++)
                        lst.add(new Integer(i));
                //*
        }
}
//exemplul 2 - List2
Adaugam la exemplul 1 (*) urmatoarea secventa
                for(int i=0; i < N; i++)
                        lst.get(i);
//exemplul 3 - List3
Adaugam la exemplul 1 (*) urmatoarea secventa
```

```
for(int i=0; i < N; i++)
        lst.remove(0);
```

Timpii aproximativi de rulare a acestor programe sunt dati in tabelul de mai jos:

	ArrayList	LinkedList
List1 (add)	0.4	0.4
List2 (get)	0.4	21.3
List3 (remove)	6.9	0.4

Asadar, adaugarea elementelor este rapida pentru ambele tipuri de liste. `ArrayList` ofera acces in timp constant la elementele sale si din acest motiv folosirea lui "get" este rapida, în timp ce pentru `LinkedList` este extrem de lenta, deoarece intr-o lista inlantuita accesul la un element se face prin parcurgerea secventiala a listei pâna la elementul respectiv.
La eliminarea elementelor din lista folosirea lui `ArrayList` este lenta deoarece elementele ramase sufera un proces de reindexare (shift la stânga) in timp ce pentru `LinkedList` este rapida si se face prin simpla schimbare a unor legaturi.
Deci, `ArrayList` se comporta bine pentru cazuri in care avem nevoie de regasirea unor elemente la pozitii diferite in lista, iar `LinkedList` functioneaza optim atunci când facem multe operatii de editare(stergeri, inserari) în corpul listei.
De asemenea, `ArrayList` foloseste mai eficient spatiul de memerie decât `LinkedList`, deoarece aceasta din urma are nevoie de o structura de date auxiliara pentru memorare unui nod. Nodurile sunt reprezentate prin instante ale unei clase interne, având forma:

```
class Entry {
        Object element;
        Entry next;
        Entry previous;
}
```

Concluzia nu este ca una din aceste clase este mai "buna" decât cealalta, ci ca exista diferente substantiale in reprezentarea si comportamentul diferitelor implementari ale colectiilor si ca alegerea unei clase pentru reprezentarea unei colectii trebuie sa se faca în functie de natura problemei ce trebuie rezolvata. Acest lucru este valabil pentru toate tipurile de colectii. De exemplu, `HashSet` si `TreeSet` sunt doua modalitati de reprezentare a multimilor. Prima se bazeaza pe folosirea unei tabele de dispersie, a doua pe folosirea unei structuri arborescente.

Algoritmi

Algorimtii polimorfici descrisi în aceasta sectiune sunt metode definite în clasa **`Collections`** care permit efectuarea unor operatii utile cum ar fi cautarea, sortarea,etc. Caracterisiticile principale ale algoritmilor sunt:

- sunt metode statice
- au un singur argument de tip colectie
- apelul lor va fi de forma **`Collections.algoritm([colectie])`**
- majoritatea opereaza pe liste dar si pe colectii arbitrare

Metodele mai importante din clasa Collections sunt date in tabelul de mai jos:

sort	Sorteaza ascendent o lista referitor la ordinea sa naturala sau la ordinea data de un

	comparator
shuffle	Amesteca elementele unei liste - opusul lui `sort`
binarySearch	Efectueaza o cautare binara a unui element într-o lista ordonata
reverse	Inverseaza ordinea elementelor dintr-o lista
fill	Populeaza o lista cu un element
copy	Copie elementele unei liste in alta
min	Returneaza minimul dintr-o colectie
max	Returneaza maximul dintr-o colectie
enumeration	Returneaza o enumerare a elementelor dintr-o colectie

Iteratori si enumerari

Enumerarile si iteratorii descriu modalitati pentru parcurgerea secventiala a unei colectii. Ei sunt descrisi de obiecte ce implementeaza interfetele **`Enumeration`**, respectiv **`Iterator`** sau **`ListIterator`**. Toate clasele care implementeaza colectii au metode ce returneaza o enumerare sau un iterator pentru parcurgerea elementelor lor. Metodele acestor interfete sunt date in tabelul de mai jos, semnificatiile lor fiind evidente:

Enumeration	Iterator	ListIterator

`boolean hasMoreElements()`
`Object nextElement()`
	`boolean hasNext()`
	`Object next()`
	`void remove()`
		`boolean hasNext(),hasPrevious()`
		`Object next(), previous()`
		`void add(Object o)  void remove()`
		`void set(Object o)`

Iteratorii simpli permit eliminarea elementului curent din colectia pe care o parcurg, cei ordonati (de tip `ListIterator`) permit si inserarea unui element la pozitia curenta, respectiv modificarea elementului curent.
Iteratorii sunt preferati enumerarilor datorita posibilitatii lor de a actiona asupra colectiei pe care o parcurg prin metode de tip `remove, add, set` dar si prin faptul ca denumirile metodelor sunt mai concise.
In exemplul de mai jos punem într-un vector numerele de la 1 la 10, le amestecam, dupa care le parcurgem element cu element folosind un iterator.

```
import java.util.*;
class TestIterator{
        public static void main(String args[]) {
                ArrayList a = new ArrayList();
                for(int i=0; i<10; i++) a.add(new Integer(i));
                Collections.shuffle(a);        //amestecam elementele colectiei
                System.out.println("Vectorul amestecat: " + a);
                System.out.println("Parcurgem vectorul element cu element:");
                System.out.println("\nvarianta 1: cu while");
                Iterator it = a.iterator();
                while (it.hasNext())
                        System.out.print(it.next() + " ");
                System.out.println("\nvarianta 2: cu for");
                for(it=a.iterator(); it.hasNext(); )
                        System.out.print(it.next() + " ");
        }
}
```

Exemplu

In exemplul de mai jos vom folosi clasa `HashMap` pentru a tine evidenta angajatilor unei companii. Vom folosi mecanismul serializarii pentru salvarea informatiilor intr-un fisier, respectiv pentru restaurarea lor.

```
//clasa Angajat
import java.io.Serializable;
class Angajat implements Serializable {
        String cod;
        String nume;
        int salar;
        public Angajat(String cod, String nume, int salar) {
                this.cod=cod;
                this.nume=nume;
                this.salar=salar;
        }
        public String toString() {
                return cod + "\t" + nume + "\t" + salar;
        }
}

//clasa Personal
import java.io.*;
import java.util.*;
class Personal implements Serializable {
        HashMap personal = new HashMap();
        String fisier=null;
        boolean salvat=false;

        void load(String fis) throws IOException{
                ObjectInputStream in=null;
                this.fisier=fis;
                try     {
                        in=new ObjectInputStream(new FileInputStream(fisier));
                        personal = (HashMap)in.readObject();
                } catch(FileNotFoundException e) {
                        System.out.println("Fisierul " + fisier + " nu exista!");
                } catch(ClassNotFoundException e) {
                        System.out.println("Eroare la incarcarea datelor!");
                }finally {
                        if (in != null)
                                in.close();
                }
        }
        void saveAs(String fis) throws IOException{
                ObjectOutputStream out=null;
                try {
                        out=new ObjectOutputStream(new FileOutputStream(fis));
                        out.writeObject(personal);
                        salvat=true;
                        System.out.println("Salvare reusita in fisierul " + fis);
                }catch(IOException e) {
                        System.out.println("Salvarea nu a reusit!");
                }finally {
                        if (out != null)
                                out.close();
                }
        }
        void save() throws IOException {
                if (fisier == null) fisier="personal.txt";
                saveAs(fisier);
```

```
        }

        Angajat getAngajat(String argumente) {
                String cod="", nume="";
                int salar=0;
                try {
                        StringTokenizer st=new StringTokenizer(argumente);
                        cod = st.nextToken();
                        nume = st.nextToken();
                        salar = Integer.parseInt(st.nextToken());
                }catch(NoSuchElementException e) {
                        System.out.println("Argumente incomplete!");
                }catch(NumberFormatException e) {
                        System.out.println("Salarul trebuie sa fie numeric!");
                }
                return new Angajat(cod, nume, salar);
        }

        boolean add(String argumente) {
                Angajat a=getAngajat(argumente);
                if (personal.containsKey(a.cod)) {
                        System.out.println("Mai exista un angajat cu acest cod!");
                        return false;
                }
                personal.put(a.cod, a);
                salvat=false;
                return true;
        }

        boolean delete(String cod) {
                if (personal.remove(cod) == null) {
                        System.out.println("Nu exista nici un angajat cu acest cod!");
                        return false;
                }
                salvat=false;
                return true;
        }

        void update(String argumente) {
                Angajat a=getAngajat(argumente);
                delete(a.cod);
                add(argumente);
        }

        void list() {
                Iterator it=personal.values().iterator();
                while (it.hasNext()) System.out.println((Angajat)(it.next()));
        }

        void executaComenzi() {
                String linie, comanda, argumente;
                try {
                        BufferedReader stdin=new BufferedReader(new
InputStreamReader(System.in));
                        while (true) {
                                linie = stdin.readLine().trim();
                                StringTokenizer st=new StringTokenizer(linie);
                                comanda=st.nextToken();
                                argumente="";
                                while (st.hasMoreTokens()) argumente += st.nextToken()
+ " ";
                                argumente=argumente.trim();

                                if (comanda.startsWith("exit")) break;
```

```
                                else if (comanda.startsWith("add")) add(argumente);
                                else if (comanda.startsWith("del")) delete(argumente);
                                else if (comanda.startsWith("update"))
update(argumente);
                                else if (comanda.startsWith("list")) list();
                                else if (comanda.startsWith("load")) load(argumente);
                                else if (comanda.startsWith("saveAs"))
saveAs(argumente);
                                else if (comanda.startsWith("save")) save();
                                else System.out.println("what ?");
                        }
                        if (!salvat) save();
                        System.out.println("bye...");
                }catch (IOException e) {
                        System.out.println("Eroare I/O:" + e);
                        e.printStackTrace();
                }
        }
}

//clasa principala GestiuneAngajati
public class GestiuneAngajati {
        public static void main(String args[]) {
                Personal p = new Personal();
                p.executaComenzi();
        }
}
```