CAPITOLUL 7

7.1. Colecţii

Colecţie reprezintă o realizare de tipuri abstracte (structuri) de date, care realizează două operaţii de bază:

• introduce un element nou în colectie;

• şterge elementul din colecţie.

Colecţiile sunt unite într-o bibliotecă de clase **java.util**, şi reprezinta containere pentru depozitarea şi manipularea obiectelor. Colectii - sunt tablouri dinamice, liste legate, arbori, seturi, stive, cozi. În clasa **Collection** sunt definite metode statice care lucrează cu toate colecţiile. Un alt mod de a lucra cu elementele colecţiei este utilizarea metodelor din clasa **Iterator**, care prevede un mijloc de parcurgere a conţinutului colectiei.

Clasele colecţiilor:

`Collection` – vârful ierarhiei a claselor;

`List` – extinderea colectiei pentru prelucrare listei;

`Set` – extinderea colecţii pentru prelucrarea mulţimei (seturi), care conţine elemente unice;

`Map` – afişarea datele sub formă de cheie-valoare .

Toate clasele de colectii realizează şi interfaţa **`Serializable`**.

Metodele clasei **`Collection`**:

`boolean add(Object obj)` – adaugă **obj** la colectia activată şi returnează **true**, dacă obiectul se adaugă, şi **false**, dacă **obj** este deja un element a colectiei. Tot asa ca **`Object`** - superclasa pentru toate clasele, într-o colecţie, se pot păstra obiecte de orice tip, cu excepţia celor de bază;

`boolean addAll(Collection c)` –adaugă toate elementele colectiei la colectia activată;

`void clear()` –sterge toate elementele din colectie;

`boolean contains(Object obj)` – returneaza **true**, daca colectia contine elemental **obj**;

`boolean equals(Object obj)` — returneaza **true,** daca colectiile sunt egale;

`boolean isEmpty()` – returnează **true,** daca colectia este goala;

`Iterator iterator()` – regăseşte un iterator;

`boolean remove(Object obj)` –şterge **obj** din colectie;

`int size()` – returnează numărul de elemente din colecţie;

`Object[] toArray()` –copie elementele colectiei intr-un tablou;

Pentru a lucra cu elementele colectiei se utilizeaza următoarele clase:

`Comparator` –pentru compararea obiectelor;

`Iterator`, **`ListIterator`**, **`Map`** – pentru a enumera şi accesa obiectele din colecţie.

Clasa **Iterator** se utilizeaza pentru accesul la elementele colectiei.

Iteratorii se plasează intre elementele din colectie.

Metodele clasei **Iterator:**

`Object next()` – returnează un obiect la care arata iteratorul, şi mută indicatorul curent la urmatorul iterator, asigură accesului la elementul următor. Dacă elementul urmator din colectie este absent, metoda **next()**, genereaza o excepţie **`NoSuchElementException`**;

`boolean hasNext()` – verifică elementul următor, şi dacă nu-i, returnează **false**. Iteratorul în acest caz rămâne neschimbat;

`void remove()` –sterge obiectul , returnat de ultimul apel a metodei **next();**

Clasa **ListIterator** extinde clcsa **Iterator** şi este destinat în primul rând pentru a lucra cu liste. Disponibilitatea metodelor **`Object previous()`**, **`int previousIndex()`** si **`boolean hasPrevious()`** prevede navigarea de la sfîrşitul listei . Metoda **`int nextIndex()`** returneaza numarul urmatorului iterator . Metoda **`add(Object ob)`** permite inserarea a unui element in listă in poziţia curentă. Apelul metodei **`void set(Object ob)`** înlocuieşte elementul listei curent cu obiectul care este transmis metodei ca parametru.

Clasa **`Map.Entry`** este predefinită pentru a extrage cheile şi valorile folosind metode **getKey()** şi **getValue()**, respectiv. Apelarea metodei **`setValue(Object value)`** înlocuieşte valoarea asociată cu cheia curenta.

7.2. Liste

Clasa **ArrayList** este un masiv dinamic de referinţe la obiecte. Extinde clasa **AbstractList** şi realizeaza interfata **List**. Clasa are următorii constructori:

`ArrayList()`
`ArrayList(Collection c)`
`ArrayList(int capacity`)

Practic, toate metodele clasei sunt o implementare a metodelor abstracte din superclasele sale şi interfeţe. Metode interfeţei **List** permite de a insera şi şterge elemente din pozitii, indicate de indice:

`void add(int index, Object obj)` – inserază **obj** în poziţia indicată de **index;**

`void addAll(int index, Collection c)` – inserează în listă toate elementele colecţiei **c**, incepand de la pozitia **index**;

`Object get(int index)` – returnează elementul în forma de obiect din poziţia **index**;

`int indexOf(Object ob)` – returnează indexul obiectului indicat;

`Object remove(int index)` –sterge obiectul din pozitia index;

Stergerea elementelor colectiei este o sarcină voluminoasă, astfel un obiect ArrayList este potrivit pentru a păstra liste neschimbate.

```
    /* exemplu # 1 : Lucru cu liste : */
import java.util.*;
public class DemoList1 {
public static void main(String[] args) {
          List c = new ArrayList();
//Collection c = new ArrayList();
//incearcă aşa!
          int i = 2, j = 5;
          c.add(new Integer(i));
          c.add(new Boolean("True"));
          c.add("<STRING>");
          c.add(2,Integer.toString(j) + "X");
```

```
                //schimbă  2 cu 5 !
          System.out.println(c);
          if (c.contains("5X"))
               c.remove(c.indexOf("5X"));
          System.out.println(c);
     }}
```

Rezultatul la consolă va fi:

```
[2, true, 5X, <STRING>]
[2, true, <STRING>]
```

Pentru a accesa elementele din listă pot fi utilizată interfaţa **ListIterator**, în timp ce clasa **ArrayList** dispune de metode similare, în special **setObject(int index, Object ob),** care permite să înlocuiască elementul din listă fără iterator, returnînd elementul şters.

```
    /* exemplu # 2 : schimbul si stergerea elementelor : */
import java.util.*;
public class DemoList2 {
     static ListIterator it;
public static void main(String[] args) {
          ArrayList a = new ArrayList();
          int index;
System.out.println("colecţia e pustie: "
               + a.isEmpty());
          Character ch = new Character('b');
          a.add(ch);
          for (char c = 'a'; c < 'h'; ++c)
               a.add(new Character(c));
System.out.println(a+"numărul de elemente:"
          + a.size());
it = a.listIterator(2);
//exstragerea iteratorului listei
     it.add("new"); //adăugarea elementului
     System.out.println(
a + "adăugarea elementului în poziţia");
System.out.println("numărul de elemente este:" + a.size());
//compararea metodelor
index = a.lastIndexOf(ch);
//index = a.indexOf(ch);
a.set(index, "rep"); //schimbul elementului
     System.out.println(a +
     "schimbul elementului");
     a.remove(6); //ştergerea elementului
     System.out.println(a +
"este şters al 6-lea element");
System.out.println("colecţia e pustie: "
          + a.isEmpty());
     }}
```

Colectia **LinkedList** pune în aplicare o listă înlănţuită. În contrast cu matricea, care stochează obiectele din locaţiile de memorie consecutive, lista legată pastreaza obiectul separat, însă

împreună cu link-urile următoarei şi anterioarei secvenţe a şirului. În lista alcătuit din N elemente, există N +1 poziţii a iteratorului.

Plus la toate metodele care există in **LinkedList** se realizeaza metodele **void addFirst(Object ob)**, **void addLast(Object ob)**, **Object getFirst()**, **Object getLast()**, **Object removeFirst()**, **Object removeLast()** care adăuga, extrage, şterge şi extrage primul şi ultimul element din listă.

```
    /* exemplu # 3 : adaugarea si stergerea elementelor : */
import java.util.*;
public class DemoLinkedList {
     public static void main(String[] args){
     LinkedList aL = new LinkedList();
     for(int i = 10; i <= 20; i++)
          aL.add("" + i);
     Iterator it = aL.iterator();
     while(it.hasNext())
     System.out.print(it.next() + " -> ");
     ListIterator list = aL.listIterator();
          list.next();
System.out.println("\n" + list.nextIndex()
               + "indice");
//ştergerea elementului cu indecele curent //list.remove();
     while(list.hasNext())
list.next();//trecerea la indicele următor
     while(list.hasPrevious())
/*exstragere în ordine inversă */
System.out.print(list.previous() + "  ");
//metodele LinkedList
aL.removeFirst();
     aL.removeLast();
     aL.removeLast();
     aL.addFirst("FIRST");
     aL.addLast("LAST");
     System.out.println("\n" + aL);
     }}
```

7.3. Set

Interfaţa **Set** declară comportamentul colectii, nu permite suprapunerea elementelor. Interfaţa **SortedSet** moşteneşte **Set** şi anunţa comportamentul mulţimii, sortate în ordine crescătoare cu metodele **first() / last(),** care returnează primul şi ultimul elemente.

Clasa **HashSet** moşteneşte de la clasa abstractă **AbstractSet** şi implementează interfaţa **Set**, utilizând un tabel **hash** pentru a stoca colectia. Cheia (cod hash) este folosit în loc de indece pentru acces la date, care accelerează mult căutarea unui anumit element. Viteza de căutare este esenţială pentru colecţiile cu un număr foarte mare de elemente. Toate elementele dintr-o mulţime sunt sortate cu ajutorul tabelului de hash, care stochează hash codurile elementelor.

Constructorii clasei:

HashSet()

HashSet(Collection c)

HashSet(int capacity)
HashSet(int capacity, float fillRatio),
unde **capacity** – numarul de celule pentru pastrarea hash-coduri.

```
/* exemplu # 4 : utilizarea mulţimii pentru exstragerea
cuvintelor unice din fişier : */
import java.util.*;
import java.io.*;
class DemoHashSet {
public static void main(String[] args) {
          Set words = new HashSet(100);
// utilizarea colecţiilor LinkedHashSet sau //TreeSet
long callTime = System.currentTimeMillis();
          try {
BufferedReader in = new BufferedReader(
new FileReader("c://pushkin.txt"));
//La sfîrşitul fişierului trebie să fie //şirul END
               String line = "";
while(!(line = in.readLine()).equals("END")) {
     StringTokenizer tokenizer =
               new StringTokenizer(line);
     while(tokenizer.hasMoreTokens()) {
     String word = tokenizer.nextToken();
               words.add(word);
                    }}
          } catch (IOException e) {
               System.out.println(e);
          }
          Iterator it = words.iterator();
          while (it.hasNext())
               System.out.println(it.next());
          long totalTime =
System.currentTimeMillis()- callTime;
System.out.println("cuvinte: " + words.size() + ", "+ totalTime +
" milisecunde");
     }}
```

Clasa **TreeSet serveşte** pentru stocarea obiectelor folosind un arbore binar, specifica căruia este sortarea elementelor sale. Atunci când este adăugatun obiect la arbore el este imediat pus în poziţia necesară, ţinând cont de sortare. Sortarea se datorează faptului că toate elementele adăugate realizează interfata **Comparable**. Operaţiile de ştergere şi inserare de obiecte sunt mai lente decât un hash-set, dar mai rapid decât în liste.

Clasa **TreeSet** conţine metode pentru a prelua prima şi ultima (cel mai mic şi cel mai mare) elemente **first()** si **last().** Metodele **SortedSet subSet(Object from, Object to)**, **SortedSet tailSet(Object from)** и **SortedSet headSet(Object to)** servesc pentru a extrage o anumită parte a setului.

```
    /* exemplu # 5: crearea mulţimelor din listă si metodele lor
*/
```

```
import java.util.*;
public class DemoTreeSet {
     public static void main(String[] args) {
          Collection c = new ArrayList();
          boolean b;
          for (int i = 0; i < 6; i++)
     c.add(Integer.toString(
          (int) (Math.random() * 90)) + 'Y');
          System.out.println(c + "lista");
          TreeSet set = new TreeSet(c);
          System.out.println(set + "mulţimea");
b = set.add("5 Element"); //adăugarea(b=true)
b = set.add("5 Element"); //adăugarea(b=false)
          //după adăugare
          System.out.println(set + "add");
          Iterator it = set.iterator();
          while (it.hasNext()) {
               if (it.next() == "5 Element")
                    it.remove();
          }
          //după ştergere
          System.out.println(set + "delete");
     //exstragerea elementului maximalşi minimal
          System.out.println(set.last() + " "
               + set.first());
     }
}
```

Rezultatul realizării:

```
[42Y, 61Y, 55Y, 3Y, 4Y, 55Y]lista
[3Y, 42Y, 4Y, 55Y, 61Y]set
[3Y, 42Y, 4Y, 5  Element, 55Y, 61Y]add
[3Y, 42Y, 4Y, 55Y, 61Y]delete
 61Y 3Y
```

Mulţimea este iniţializată de listă şi sortată imediat în procesul de creare . După adăugarea unui nou element a făcut o încercare nereuşită de a se adăuga din nou. Folosind iteratorul elementul elementul poate fi găsit şi scos din mulţime.

7.4. Hărţi Map

Harta **Map** - este un obiect care deţine o pereche de cheie-valoare. Cautarea obiectului (valori) este facilitată în comparaţie cu mulţimile, datorită faptului că aceasta poate fi găsit după cheie individuală. În cazul în care elementul cu cheia specificată nu este găsit, atunci este returnat null.

Clasele de hărţi :

AbstractMap – realizeaza interfaţa Map;

HashMap – extinde **AbstractMap**,utilizind hash-tabel, în care cheile sunt ordonate în dependenţă de valorile codurilor hash;

TreeMap –extinde **AbstractMap**, folosind un arbore în care cheile sunt aranjate într-un arbore de căutare într-un mod ordonat.

Interfetele harţii:

Map – afişează chei şi valorile unice;

Map.Entry – descrie o pereche de cheie-valoare

SortedMap – conţine cheile sortate.

Interfata **Map** contine urmatoarele metode:

void clear() –elimină toate perechile din hartă;

boolean containsKey(Object obj) – returneaza **true**, daca harta chemata contine **obj** ca cheie;

Set entrySet() – returnează un set care conţine valorile harţii;

Set keySet() –returnează un set de cheii;

Object get(Object obj) – returnează valoarea asociată cu cheia **obj;**

Object put(Object obj1, Object obj2) – pune cheia **obj1** şi valoarea **obj2** in hartă. Când este adăugat elementul in hartă cu o cheie existentă va fi înlocuit elementul curent cu cel nou. Metoda va returna elementul înlocuit;

Collection values() –returneaza o colectie ce conţine conţinutul harţi;

Interfata **Map.Entry** contine urmatoarele metode:

Object getKey() – returnează cheia curentă;

Object getValue() – returnează valoarea curentă;

Object setValue(bject obj) – stabileşte valoarea **obj** în poziţia curentă.

Exemplul de mai jos arată cum de creat un hash-hartă şi acces la elementele sale.

```
    /* exemplu # 6 : crearea hash-hărţii şi schimbarea elementului
după cheie:  */
import java.util.*;
public class DemoHashMap {
     public static void main(String[] args){
          Map hm = new HashMap(5);
          for (int i = 1; i < 10; i++)
     hm.put(Integer.toString(i), i + " element");
          hm.put("14s", new Double(1.01f));
          System.out.println(hm);
          hm.put("5", "NEW");
System.out.println(hm + "modificarea elementului ");
          Object a = hm.get("5");
System.out.println(a + " - a fost găsit după cheie '5'");
/* exstregerea hash-tabelului cu metoda interfeţii Map.Entry */
          Set set = hm.entrySet();
          Iterator i = set.iterator();
          while(i.hasNext()){
          Map.Entry me = (Map.Entry)i.next();
```

```
            System.out.print(me.getKey()+" : ");
            System.out.println(me.getValue());
            }
      }
}
```

Mai jos urmează un fragment de sistem corporativ, care demonstrează posibilitatea clasei **HashMap** şi a interfeţii **Map.Entry** la stabilirea drepturilor utilizatorilor.

```
    /* exemplu # 7 : utilizarea colecţiilor pentru controlul
accesului la sistem : */
import java.util.*;
public class DemoSecurity {
     public static void main(String[] args) {
     CheckRight.startUsing("2041", "Bill G.");
     CheckRight.startUsing("2420", "George B.");
/*adăugarea unui nou utilizator controlul nivelului de acces */
     CheckRight.startUsing("2437", "Phillip K.");
     CheckRight.startUsing("2041", "Bill G.");
     }
}
class CheckRight {
     private static HashMap map = new HashMap();
     public static void startUsing(
                    String id, String name) {
          if (canUse(id)){
               map.put(id, name);
          System.out.println("accesul este acordat");
          }
          else {
          System.out.println("accesul este interzis");
          }
     }
     public static boolean canUse(String id) {
          final int MAX_NUM = 2;//de schimbat 2 pe 3
          int currNum = 0;
          if (!map.containsKey(id))
                currNum = map.size();
          return currNum < MAX_NUM;
          }
     }
```

Rezultatul realizăriit:
accesul este acordat,
accesul este acordat
accesul este interzis
accesul este interzis,

Astfel încât să aibă acces la sistem simultan permis doar pentru doi utilizatori. În cazul în care în codul programului se va modifica valoarea constantei MAX_NUM mai mult de 2, atunci noul utilizator primeste drepturi de acces.

Clasa **WeakHashMap** permite mecanizmului de distrugere a obiectelor de a stergere din hartă valoarea dupa cheie, referinţă căruia a ieşit din domeniul de aplicare al programului.

Clasa **LinkedHashMap** memorează lista obiectelor adăugate la hartă şi formează o listă dublu-înlănţuită. Acest mecanism este eficient doar dacă este supraîncărcat coeficientul hărţii, atunci când se lucrează cu cache-memorie.

Începând cu versiunea de Java 1. 4 a fost adăugată clasa **IdentityHashMap**, codurile hash a obiectelor-cheie sunt calculate de metoda **System.identityHashCode()** după adresa obiectului în memorie, în contrast cu metoda **hashCode (),** care calculate exclusiv pe conţinutul obiectului.

7.5. Colecţii moştenite

În unele tehnologii, cum ar fi Servletele, pina acum inca se folesesc colectiile care existau în Java iniţial, şi anume harta **Hashtable** şi enumerarea **Enumaration.**

```
    /* exemplu # 8 : crearea hash-tabel şi căutarea elementelor
după cheie :  */
import java.util.*;
import java.io.*;
public class HashTableDemo {
     public static void main(String[] args) {
          Hashtable capitals = new Hashtable();
          showAll(capitals);
          capitals.put("Ucraina", "Kiev");
          capitals.put("Franţa", "Paris");
          capitals.put("Belarusi", "Minsc");
          showAll(capitals);
          //căutarea după cheie
     System.out.print("întroduceţi ţara: ");
          BufferedReader br =
             new BufferedReader(
               new InputStreamReader(System.in));
          String name;
          try {
               name = br.readLine();
               showCapital(capitals, name);
          } catch (IOException e) {
               e.printStackTrace();
          }
     }
     private static void showAll(Hashtable capitals){
          Enumeration countries;
          if (capitals.isEmpty())
               System.out.println("tabelul este gol");
          else {
```

```
                countries = capitals.keys();
                String country;
           while (countries.hasMoreElements()) {
      country = (String) countries.nextElement();
                System.out.print(country + " - ");
           System.out.println(capitals.get(country));
                }
           }
      }
      private static void showCapital(
                Hashtable cap, String country) {
           if (cap.get(country) != null) {
                System.out.print(country + " - ");
                System.out.println(cap.get(country));
           } else
           System.out.println("înscrierea lipseşte");
      }
}
```

Ca rezultat, la consolă se va afişa:
tabelul este gol
Belarus - Minsk
Franţa - Paris,
Ucraina - Kiev,
introduceţi ţară: Ucraina
Ucraina - Kiev

```
    exemplu # 9 : crearea colecţiei parametrizate :  */
import java.util.*;
```

public class Principiul de lucru cu colectiile, în comparaţie cu structura lor, la schimbare versiuni limbajilui Java nu s-au schimbat în mod semnificativ.

7.6. Parametrizarea colectiei

Este propus un mecanism mult mai convenabil pentru lucrul cu colectiile, şi anume:

• nu este nevoie în mod constant pentru a converti obiectele returnate (de tip Object) la tipul dorit;

• compilatorului se transmite un raport preliminar cu privire la tipul de obiecte care vor fi stocate în colectarea şi verificarea se face la compilare.

```
/* DemoGenerics {
  public static void main(String args[]) {
    Map <String, Integer> map =
                new HashMap <String, Integer> ();
    map.put("Key 1", 1);
    int res = map.get("Key 1");/* compilatorul cunoaşte tipul
valorii */
Character ch = new Character('2');
```

```
//  map.put(ch, 2);//errore de compilare
//compilatorul nu permite de a adăuga un alt tip
 }
}
```

În această situaţie nu creează o nouă clasă pentru fiecare tip şi colectia nu este schimbată, pur şi simplu compilatorul conţine informaţii cu privire la tipul de elemente care pot fi stocate în hartă. Parametrul colectiei nu poate fi un tipul de bază.

Trebuie remarcat faptul că ar trebui să indice tipul la crearea unei referinţe, altfel va fi permis pentru a adăuga obiecte de toate tipurile.

```
// exemplu # 10 : parametrizarea :
import java.util.*;
public class UncheckCheck {
  public static void main(String args[]) {
    Collection c1 = new HashSet <String> ();
    c1.add("Java");
    c1.add(5); //nu este error: c1 nu este parametrizat
    for(Object ob : c1)
      System.out.print(ob);
    Collection <String> c2 = new HashSet<String>();
    c2.add("A");
//  c2.add(5);
//error de compilare: deoarece c2 ecte parametrizat
    }
}
```

Rezultatul realizării va afisa :

Java5

Pentru ca parametrizarea colectiei sa fie completă, trebuie de specificat un parametru şi atunci când se declară un link, şi atunci când se declară un obiect.

Există biblioteci deja gata, în care nus teste de tip, prin urmare, utilizarea lor nu poate garanta că in colectie nu va fi plasat un obiect de alt tip. Pentru acesta în clasa **Collections** afost adăugat[o nouă metodă **checkedCollection()**:

```
    public static < E > Collection < E >
    checkedCollection(Collection< E > c, Class< E > type)
```

Această metodă creează o colecţie, verificaţă în faza de implementare, de exemplu, în cazul de adăugare a unui obiect de alt tip genereaza o excepţie **ClassCastException**:

```
    /* exemplu # 11 : colecţia verificată :  */
import java.util.*;
public class SafeCollection{
  public static void main(String args[]) {
     Collection c = Collections.checkedCollection(
     new HashSet <String>(), String.class);
     c.add("Java");
     c.add(5.0);    //error
  }
}
```

În această clasă au fost adăugate o serie de metode, specializate pentru testarea anumitor tipuri de colecţii, şi anume: **`checkedList()`**, **`checkedSortedMap()`**, **`checkedMap()`**, **`checkedSortedSet()`**, **`checkedSet()`**.

În versiunea Java 5.0 au fost adăugate mai multe clase şi interfeţe noi, cum ar fi **EnumSet, EnumMap, PriorityQueue** etc. Ca o ilustrare a posibilităţilor putem considera una dintre ele - interfata **Queue:**

```
public interface Queue < E > extends Collection < E >
```

Metodele interfetei **Queue:**

Eelement() – returnează, dar nu elimina elemental din capul cozii;

boolean offer(E o) – introduce un element în coadă, dacă este posibil (de exemplu: dimensiuni limitate);

E peek() - returnează, dar nu elimina elemental din capul cozii, returnează null, în cazul în care coada este goală;

E poll() – întoarce şi elimină elemental din capul cozii, returnează null, în cazul în care coada este goală;

E remove() - retunează şi elimină elementul capul cozii .

Metode de **element()** şi **remove()** diferă de metoda **Peek()** şi **poll(),** care aruncă o excepţie în cazul în care coada este goală.

Este de remarcat faptul că clasa **LinkedList** acum în afară de interfaţă **List <E>** pune în aplicare şi **Queue:**

```
    /* exemplu # 12 : colecţia verificată :  */
import java.util.*;
public class DemoQueue {
     public static void main(String args[]) {
     LinkedList <Integer> c =
          new LinkedList <Integer> ();
     //adăugare a 10 elemente
     for (int i = 0; i < 10; i++)
               c.add(i);
     Queue <Integer> queue = c;
     for (int i : queue) //exstragerea elementelor
          System.out.print(i + " ");
          System.out.println(" :size= "
               + queue.size());
     //eliminare a 10 elemente
     for (int i = 0; i < 9; i++) {
          int res = queue.poll();
               }
     System.out.print("size= " + queue.size());
     }
}
```

In rezultatul realizării se va afisa :

```
0 1 2 3 4 5 6 7 8 9 :size=10
size=1
```

7.7. Prelucrarea tablourilor

In biblioteca **java.util** este clasa **Arrays**, care conţine metode de manipulare cu conţinutul matricei, şi anume, de a căuta, complecta, compara, converti în colectie:

`int binarySearch(``parametrii`**`)`** – modă supraîncărcată şi serveşte pentru organizarea binară de căutare într-o matrice de tipuri primitive şi obiect. Returnează poziţia primei coincidenţe;

`void fill(``parametrii`**`)`** – metodă supraîncărcată serveşte pentru complectarea tablouri de diferite tipuri şi primitive;

`void sort(``parametrii`**`)`** – metoda metodă supraîncărcată sreveşte pentru a sorta o matrice sau o parte din ea folosind interfata **Comparator** şi fără ea;

`List asList(Object [] a)` – metodă care copii elementele matrice într-un obiect de tip **List.**

Aplicarea acestor metode are loc în exemplu următor.

```
    /* Exemplu # 13 : metodele clasei Arrays : */
import java.util.*;
public class ArraysEqualDemo {
     public static void main(String[] args) {
     char m1[] = new char[3],
          m2[] = { 'a', 'b', 'c' }, i;
     Arrays.fill(m1, 'a');
     System.out.print("tabloul m1:");
     for (i = 0; i < 3; i++)
          System.out.print(" " + m1[i]);
     m1[1] = 'b';
     m1[2] = 'c';
//m1[2]='x'; //va aduce la alt rezultat
     if (Arrays.equals(m1, m2))
     System.out.print("\nm1 şi m2 sunt identice ");
     else
     System.out.print("\nm1 и m2 nu sunt identice ");
     m1[0] = 'z';
     Arrays.sort(m1);
     System.out.print("\n tabloul m1:");
     for (i = 0; i < 3; i++)
          System.out.print(" " + m1[i]);
     System.out.print(
     "\n valoarea 'c' se află pe poziţia -"
     + Arrays.binarySearch(m1, 'c'));
     }
}
```

În rezultatul realizării va fi exstras:

matricea M1: AAA

M1 şi M2 sunt identice

M1 şi M2 nu sunt identice

Valoarea ‘c’ se află pe poziţia 1

Lucrare de laborator nr. 7

1. Tema lucrării:
Crearea și parcurgerea colecțiilor

2. Scopul lucrării:

- Însuşirea modalităţilor de creare, realizare şi parcurgerea a colecţiilor în Java;

3. Etapele de realizare:

1) Crearea colecţiilor şi a hărţilor;
2) Metode de realizare a colecţiilor şi a hărţilor;
3) Metode de complectare şi exstragere a obiectelor din colecţii şi hărţi;
4) Crearea interfeţii programului;
5) Prezentarea lucrării.

4. Exemplu de realizare:

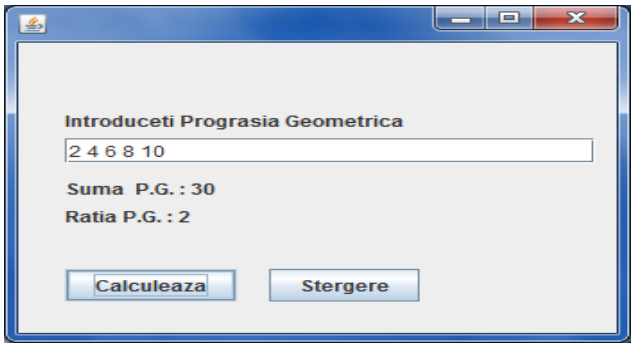
```
import java.awt.Color;
import java.util.HashSet;
import java.util.Set;
public class JFrame extends javax.swing.JFrame {
    public JFrame() {
        initComponents();
    }
    @SuppressWarnings("unchecked")
    private void initComponents() {

        jTextField1 = new javax.swing.JTextField();
        jLabel1 = new javax.swing.JLabel();
        jLabel2 = new javax.swing.JLabel();
        jLabel3 = new javax.swing.JLabel();
        jLabel4 = new javax.swing.JLabel();
        jButton1 = new javax.swing.JButton();
        jButton2 = new javax.swing.JButton();

setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOS
E);
        jTextField1.setText("1 3 5 7");
        jLabel1.setText("Introduceti Prograsia Geometrica");
        jLabel3.setText("Suma  P.G.");
        jLabel4.setText("Ratia P.G. ");
        jButton1.setText("Calculeaza");
        jButton1.addMouseListener(new
java.awt.event.MouseAdapter() {
    public void mouseClicked(java.awt.event.MouseEvent evt) {
                jButton1MouseClicked(evt);
            } });
        jButton2.setText("Stergere");
```

```
        jButton2.addMouseListener(new
java.awt.event.MouseAdapter() {
       public void mouseClicked(java.awt.event.MouseEvent evt) {
                jButton2MouseClicked(evt);
            }});
 javax.swing.GroupLayout layout =
new javax.swing.GroupLayout(getContentPane());
        getContentPane().setLayout(layout);
layout.setHorizontalGroup(layout.createParallelGroup(
javax.swing.GroupLayout.Alignment.LEADING).addGroup(
layout.createSequentialGroup().addGap(26, 26, 26).addGroup(
layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEAD
ING, false).addComponent(jLabel2,
javax.swing.GroupLayout.DEFAULT_SIZE, 296, Short.MAX_VALUE)
.addComponent(jLabel1).addComponent(jTextField1,
javax.swing.GroupLayout.DEFAULT_SIZE, 296, Short.MAX_VALUE)
.addGroup(layout.createSequentialGroup().addComponent(jButton1)
.addGap(18, 18, 18).addComponent(jButton2))
.addComponent(jLabel4, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
.addComponent(jLabel3, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
                .addContainerGap()) );
layout.setVerticalGroup(layout.createParallelGroup(
javax.swing.GroupLayout.Alignment.LEADING)
.addGroup(layout.createSequentialGroup().addGap(54, 54, 54)
.addComponent(jLabel1).addPreferredGap(
javax.swing.LayoutStyle.ComponentPlacement.RELATED)
.addComponent(jTextField1,
javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE).addPreferredGap(
javax.swing.LayoutStyle.ComponentPlacement.RELATED)
.addComponent(jLabel2).addPreferredGap(
javax.swing.LayoutStyle.ComponentPlacement.RELATED)
.addComponent(jLabel3).addPreferredGap(
javax.swing.LayoutStyle.ComponentPlacement.RELATED)
.addComponent(jLabel4).addGap(34, 34, 34).addGroup(
layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASE
LINE).addComponent(jButton1).addComponent(jButton2))
.addContainerGap(25, Short.MAX_VALUE)));
        pack();
    }
private void jButton1MouseClicked(java.awt.event.MouseEvent evt)
{
   jTextField1.setBackground(Color.white);
   jLabel3.setText("Suma  P.G. :");
```

```
   jLabel4.setText("Ratia P.G. :");
   jLabel2.setText("");
   Set s = new HashSet();
   String str[] = jTextField1.getText().split(" ");
   for(int i=0;i<str.length;i++){
       try {
          s.add(Integer.parseInt(str[i]));
       } catch (NumberFormatException numberFormatException) {
           jTextField1.setBackground(Color.red);
     jLabel2.setText("Erorr , Nu ati introdus carectere !");
       }}
   // scoatem stingul cu caractere
   String stmp = ""+s.toString();
   String tmp[] = jTextField1.getText().split(" ");
if(tmp.length >=3){
  if((Integer.parseInt(tmp[1])-
Integer.parseInt(tmp[0])+Integer.parseInt(tmp[1])) ==
Integer.parseInt(tmp[2])){
    jLabel3.setText("Suma  P.G. : "+s.hashCode());
    jLabel4.setText("Ratia P.G. : "+(Integer.parseInt(tmp[0])-
Integer.parseInt(tmp[1])+Integer.parseInt(tmp[1])));
  }
  else{
     jLabel2.setText("Nu este o progresie geometrica");
     jLabel3.setText("");
     jLabel4.setText("");
  }}}
private void jButton2MouseClicked(java.awt.event.MouseEvent evt)
{
   jTextField1.setBackground(Color.white);
   jTextField1.setText("");
   jLabel3.setText("Suma  P.G. :");
   jLabel4.setText("Ratia P.G. :");
   jLabel2.setText("");
    }
    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new JFrame().setVisible(true);
            }});}
    private javax.swing.JButton jButton1;
    private javax.swing.JButton jButton2;
    private javax.swing.JLabel jLabel1;
    private javax.swing.JLabel jLabel2;
    private javax.swing.JLabel jLabel3;
    private javax.swing.JLabel jLabel4;
    private javax.swing.JTextField jTextField1;
```

}

Rezultatul realizării programului:



5. Probleme propuse:

De realizat sarcinile utilizînd colecţiile

1. Creaţi o clasă Stack bazată pe colecţii. Introduceţi o secvenţă de caractere şi să îl imprimaţi în ordine inversă.
2. Creaţi o clasă Queue bazată pe colecţii. Introduce o serie de siruri de caractere şi stabili dacă există un şir-model în această coadă.
3. Defini o clasă Set bazată colecţii pentru un set de numere întregi, Creaţi metodele de determinare a uniunii şi intersectiei de seturi .
4. Construiţi o matrice de tip double, pe baza de colectii. Extrageţi elementele în formă: Vector: 2.3 5.0 7.3.
5. Listele I (1. . N) şi U (1. . N), conţin rezultatele măsurătorilor de tensiune şi curent pentru o rezistenţă necunoscută R. Găsiţi numărul aproximativ a rezistenţii R.
6. Efectuaţi o sumare în pereche pentru orice secvenţă finită de numere construite pe baza de colectii, după cum urmează: în prima etapă, se adună perechi de numere, la a doua etapă, se sumează perechi de rezultate a primei etape şi a.m.d. până când rămîne un rezultat. Dacă la sfîrşitul etapei rămîne număr fără pereche, el trece în etapa următoare.
7. Adunaţi două polinoame de grad fix, în cazul în care coeficienţii polinoamelor sunt stocate în obiectul HashMap.
8. Înmulţiţi două polinoame de grad fix, în cazul în care coeficienţii de polinoame sunt stocate în List.
9. Nu utilizaţi facilităţi conexe, rearanjaţi elementele negative ale listei la sfirsit, dar cele pozitive – la inceputul listei.
10. De apreciat progresia geometrică pentru o mulţime care se păstrează ăn Set.
11. De apreciat progresia aritmetică pentru un sir de numere, care se păstrează ăn List.
12. De apreciat numerele maximal şi minimal a sirului de numere care se păstrează în HashSet.