



Colecții

Programare Orientată pe Obiecte



Colecții

- Ce sunt colecțiile ?
- Interfețe ce descriu colecții
- Implementări ale colecțiilor
- Folosirea eficientă a colecțiilor
- Algoritmi polimorfici
- Tipuri generice
- Iteratori și enumerări

Ce sunt colecțiile ?

- | O colecție este un obiect care grupează mai multe elemente într-o singură unitate.
- | Tipul de date al elementelor dintr-o colecție este **Object**.

Tipuri de date reprezentate:

- vectori
- liste înlănțuite
- stive
- mulțimi matematice
- tabele de dispersie
- dicționare, etc.

Arhitectura colecțiilor

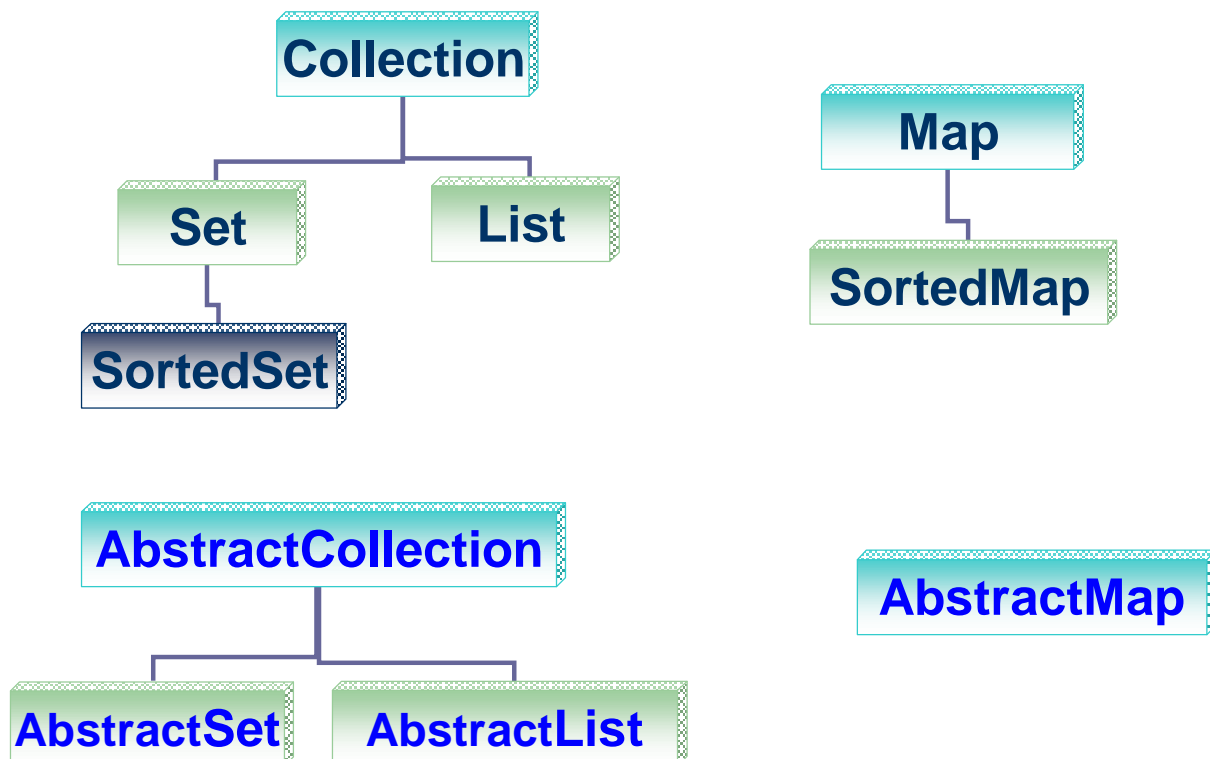
- **Interfețe:** *tipuri abstracte de date* ce descriu colecțiile și permit utilizarea lor independent de detaliile implementărilor.
- **Implementări:** implementări concrete ale interfețelor ce descriu colecții (clase); reprezintă *tipuri de date reutilizabile*.
- **Algoritmi polimorfici:** metode care efectuează diverse operații utile (căutare, sortare) definite pentru obiecte ce implementează interfețele ce descriu colecții. Acești algoritmi se numesc și polimorfici deoarece pot fi folosiți pe implementări diferite ale unei colecții, reprezentând elementul de *funcționalitate reutilizabilă*.

Avantaje:

- Reducerea efortului de programare
- Creșterea vitezei și calității programului

Interfețe ce descriu colecții

- | Collection - modelează o colecție la nivelul cel mai general, descriind un grup de obiecte (elementele sale).
- | Map - descrie structuri de date de tip cheie – valoare, asociază fiecarui element o cheie unică, după care poate fi regăsit.

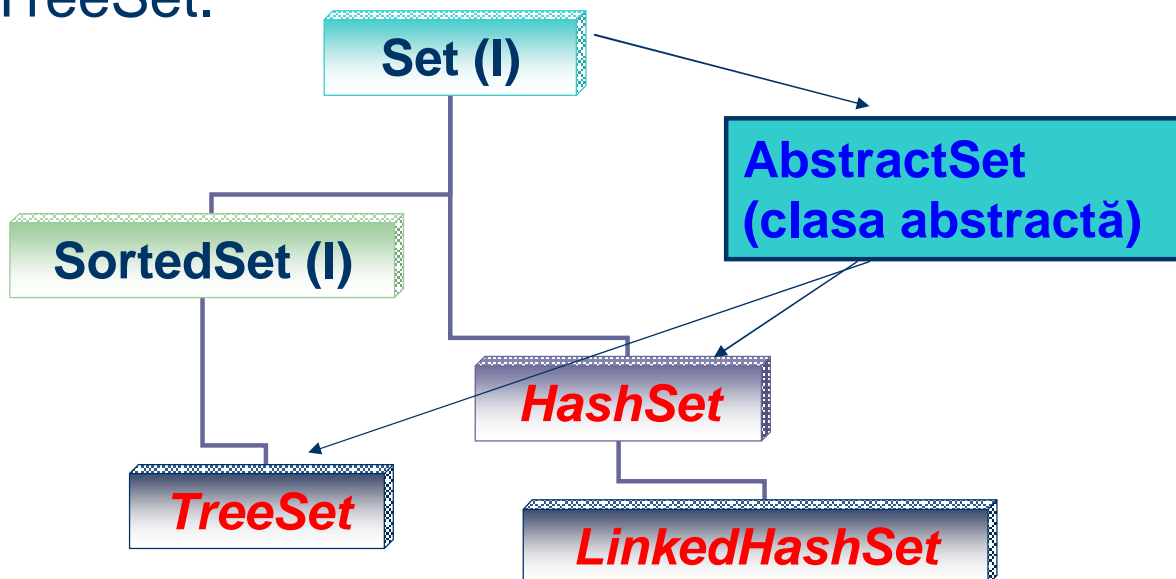


Collection

```
public interface Collection {  
    // Metode cu caracter general  
    int size();  
    boolean isEmpty();  
    void clear();  
    Iterator iterator();  
    // Operatii la nivel de element  
    boolean contains(Object element);  
    boolean add(Object element);  
    boolean remove(Object element);  
    // Operatii la nivel de multime  
    boolean containsAll(Collection c);  
    boolean addAll(Collection c);  
    boolean removeAll(Collection c);  
    boolean retainAll(Collection c);  
    // Metode de conversie in vector  
    Object[] toArray();  
    ...  
}
```

Set

- | Mulțime în sens matematic.
- | O mulțime nu poate avea elemente duplicate: nu poate conține două obiecte $o1$ și $o2$ cu proprietatea $o1.equals(o2)$.
- | Implementări: `HashSet`, `LinkedHashSet` și `TreeSet`.



SortedSet

- I mulțime cu elemente sortate.
- I ordonarea elementelor se face conform **ordinii lor naturale**, sau conform cu **ordinea dată de un comparator specificat** la crearea colecției
- I ordinea este menținută automat la orice operație efectuată asupra mulțimii.
- I pentru orice două obiecte o1, o2 ale colecției, apelul o1.compareTo(o2) (sau comparator.compare(o1, o2), dacă este folosit un comparator) trebuie să fie valid și să nu provoace excepții

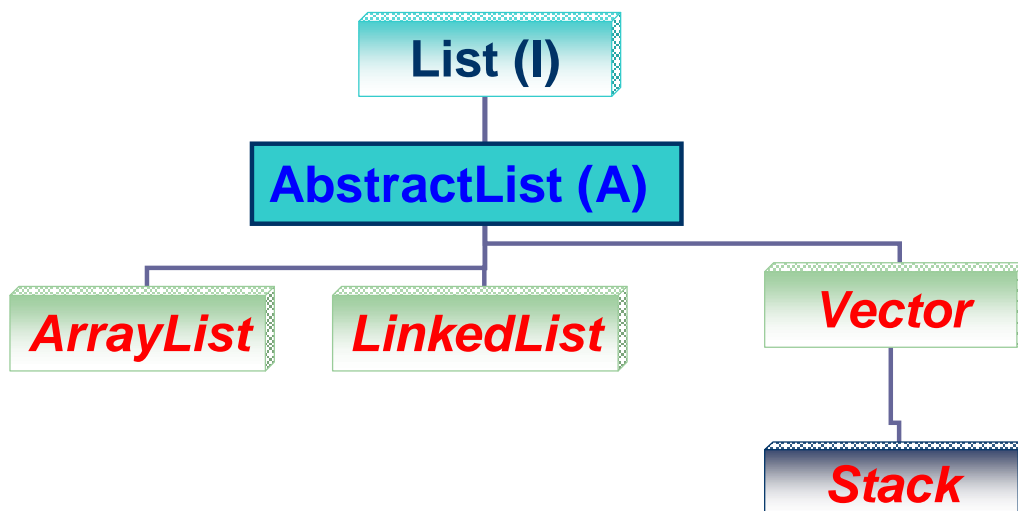
SortedSet

```
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();
}
```

Implementare: **TreeSet** – help

List

- | liste de elemente indexate.
- | pot contine duplicate și permit un control precis asupra poziției unui element prin intermediul indexului acelu element.
- | există metode pentru acces pozițional, căutare și iterare avansată.
- | Implementări: ArrayList, LinkedList, Vector, Stack.



List (2)

```
public interface List extends Collection {  
    // Acces pozitional  
    Object get(int index);  
    Object set(int index, Object element);  
    void add(int index, Object element);  
    Object remove(int index);  
    abstract boolean addAll(int index, Collection c);  
    // Cautare  
    int indexOf(Object o);  
    int lastIndexOf(Object o);  
    // Iterare  
    ListIterator listIterator();  
    ListIterator listIterator(int index);  
    // Extragere sublista  
    List subList(int from, int to);  
}
```

Map

- I structuri de tip: cheie – element, ce asociază fiecarui element o cheie unică, după care poate fi regăsit.
- I nu pot conține chei duplicate și fiecare cheie este asociată unui singur element.

```
public interface Map {  
    // Metode cu caracter general  
    int size();  
    boolean isEmpty();  
    void clear();  
    // Operatii 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);  
}
```

Map(2)

```
// Operatii la nivel de multime
void putAll(Map t);
// 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);
}
}
```

Implementări: **HashMap**, **TreeMap** și **Hashtable**.

SortedMap

- | Mulțimea cheilor este sortată conform ordinii naturale sau unui comparator.

```
public interface SortedMap extends Map {  
    // Extragerea de subtabele  
    SortedMap subMap(Object fromKey, Object  
        toKey);  
    SortedMap headMap(Object toKey);  
    SortedMap tailMap(Object fromKey);  
    // Capete  
    Object first();  
    Object last();  
    // Comparatorul folosit pentru ordonare  
    Comparator comparator();  
}
```

- | Implementare: **TreeMap** - help

Implementări ale colecțiilor

< Implementare >< Interfața >

Interfața	Clasa
Set	HashSet, <i>LinkedHashSet</i>
SortedSet	TreeSet
List	ArrayList, LinkedList, Vector, <i>Stack</i>
Map	HashMap, Hashtable
SortedMap	TreeMap

| Organizare ierarhică

AbstractCollection - AbstractSet, AbstractList -
HashSet, TreeSet... Vector-Stack

AbstractMap - HashMap, TreeMap, Hashtable

In vechea ierarhie:

Dictionary - Hashtable – Properties

| se observă existența unor clase care oferă aceeași funcționalitate, cum ar fi ArrayList și Vector, HashMap și Hashtable.

Caracteristici comune

- permit elementul null
- sunt serializabile
- au definită metoda clone
- au definită metoda toString
- permit crearea de iteratori
- au atât constructor fără argumente cât și un constructor care acceptă ca argument o altă colecție
- exceptând clasele din arhitectura veche, nu sunt sincronizate.

Folosirea eficientă a colecțiilor

| ArrayList sau LinkedList ?

```
import java . util . * ;
public class TestEficienta {
    final static int N = 100000;
    public static void testAdd ( List lst) {
        long t1 = System . currentTimeMillis ();
        for (int i=0; i < N; i++)
            lst.add (new Integer (i));
        long t2 = System . currentTimeMillis ();
        System . out. println ("Add: " + (t2 - t1));
    }
    public static void testGet ( List lst) {
        long t1 = System . currentTimeMillis ();
        for (int i=0; i < N; i++)
            lst.get (i);
        long t2 = System . currentTimeMillis ();
        System . out. println ("Get: " + (t2 - t1));
    }
}
```

ArrayList sau LinkedList ?

```
public static void testRemove ( List lst ) {
    long t1 = System . currentTimeMillis ();
    for (int i=0; i < N; i++)
        lst. remove (0);
    long t2 = System . currentTimeMillis ();
    System . out. println (" Remove : " + (t2 - t1));
}
public static void main ( String args []) {
    System . out. println (" ArrayList ");
    List lst1 = new ArrayList ();
    testAdd ( lst1 );
    testGet ( lst1 );
    testRemove ( lst1 );
    System . out. println (" LinkedList ");
    List lst2 = new LinkedList ();
    testAdd ( lst2 );
    testGet ( lst2 );
    testRemove ( lst2 );
}
}
```

ArrayList sau LinkedList ?

	ArrayList	LinkedList
add	0.12	0.14
get	0.01	87.45
remove	12.05	0.01

Concluzii:

- | există diferențe substanțiale în reprezentarea și comportamentul diferitelor implementări
- | alegerea unei anumite clase pentru reprezentarea unei mulțimi de elemente trebuie să se facă în funcție de natura problemei ce trebuie rezolvată.

Algoritmi polimorfici

I Metode definite în clasa **Collections**:
căutare, sortare, etc.

Caracteristici comune:

- sunt metode de clasă (statice);
- au un singur argument de tip colecție;
- apelul lor general va fi de forma:

Collections.algorithm(colectie,[argumente]);

- majoritatea operează pe liste dar și pe colecții arbitrare.

Exemple de algoritmi

- sort
- shuffle
- binarySearch
- reverse
- fill
- copy
- min
- max
- swap
- enumeration
- unmodifiableTipColectie
- synchronizedTipColectie

Tipuri generice

I Tipizarea elementelor unei colecții:

TipColecție < TipDeDate >

// Inainte de 1.5

```
ArrayList list = new ArrayList();  
list.add(new Integer(123));  
int val = ((Integer)list.get(0)).intValue();
```

// Dupa 1.5, folosind tipuri generice

```
ArrayList<Integer> list = new ArrayList<Integer>();  
list.add(new Integer(123));  
int val = list.get(0).intValue();
```

// Dupa 1.5, folosind si autoboxing

```
ArrayList<Integer> list = new ArrayList<Integer>();  
list.add(123);  
int val = list.get(0);
```

I Avantaje: simplitate, control (eroare la compilare vs. ClassCastException)

Iteratori și enumerări

- | Parcurgerea secvențială a unei colecții, indiferent dacă este indexată sau nu.
- | Toate clasele care implementează colecții au metode ce returnează o enumerare sau un iterator pentru parcurgerea elementelor lor.
- | Deoarece funcționalitatea interfeței Enumeration se regăsește în Iterator, aceasta din urmă este preferată în noile implementări ale colecțiilor.

Enumeration:

// Parcurgerea elementelor unui vector v

```
Enumeration e = v.elements();
```

```
while (e.hasMoreElements()) {  
    System.out.println(e.nextElement());  
}
```

// sau, varianta mai concisa

```
for(Enumeration e=v.elements(); e.hasMoreElements();){  
    System.out.println(e.nextElement());  
}
```

Iterator, ListIterator

| **Iterator**: hasNext, next, remove

// Parcurgerea elementelor unui vector si eliminarea elementelor nule

```
for (Iterator it = v.iterator(); it.hasNext();) {  
    Object obj = it.next();  
    if (obj == null) it.remove();  
}
```

| **ListIterator**: hasNext, hasPrevious, next, previous, remove, add, set

// Parcurgerea elementelor unui vector si inlocuirea elementelor nule cu 0

```
for (ListIterator it = v.listIterator(); it.hasNext();) {  
    Object obj = it.next();  
    if (obj == null) it.set(new Integer(0));  
}
```


Folosirea unui iterator

```
import java . util . * ;
class TestIterator {
    public static void main ( String args [] ) {
        ArrayList a = new ArrayList ( ) ;
        // Adaugam numerele de la 1 la 10
        for ( int i = 1 ; i <= 10 ; i ++ ) a . add ( new Integer ( i ) ) ;
        // Amestecam elementele colectiei
        Collections . shuffle ( a ) ;
        System . out . println ( " Vectorul amestecat : " + a ) ;
        // Parcurgem vectorul
        for ( ListIterator it = a . listIterator ( ) ; it . hasNext ( ) ; ) {
            Integer x = ( Integer ) it . next ( ) ;
            // Daca elementul curent este par , il facem 0
            if ( x . intValue ( ) % 2 == 0 ) it . set ( new Integer ( 0 ) ) ;
        }
        System . out . print ( " Rezultat : " + a ) ;
    }
}
```

Varianta simplificată (1.5)

Atenție!

- I Deoarece colecțiile sunt construite peste tipul de date Object, metodele de tip *next* sau *prev* ale iteratorilor vor returna tipul Object, fiind responsabilitatea noastră de a face conversie (cast) la alte tipuri de date, dacă este cazul.

```
ArrayList<Integer> list=new ArrayList<Integer>();  
for (Iterator i = list.iterator(); i.hasNext();) {  
    Integer val=(Integer)i.next();  
    // Proceseaza val  
    ...  
}
```

sau:

```
ArrayList<Integer> list=new ArrayList<Integer>();  
for (Integer val : list) {  
    // Proceseaza val  
    ...  
}
```

Exemplu

1. Să se definească o clasă "MyArray" pentru vectori de obiecte care nu se extind dinamic și o clasă iterator pe acest vector care să implementeze interfața "Enumeration". Program pentru crearea unui obiect "MyArray" prin adăugări succesive de șiruri și afișarea lui folosind un obiect iterator. Clasa "MyArray" are un constructor cu argument întreg (dimensiune vector) și metodele:
 - add(Object): adăugare obiect la sfârșit
 - elements(): creare iterator pentru acest vector
 - get(int): obiect dintr-o poziție data
 - size(): dimensiune vector
 - toString: șir cu elementele din vector.

Rezolvare

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

```
class MyArray {
```

```
    private Object v[];
```

```
    private int n,nmax;
```

```
    public MyArray (int nmax) {
```

```
        this.nmax=nmax;
```

```
        n=0;
```

```
        v= new Object[nmax];
```

```
    }
```

```
    public void add (Object e) {
```

Rezolvare

```
public Enumeration elements() {  
    return new ArrayEnum(this);  
}
```

```
public String toString () {  
    String s="";  
    for (int i=0;i<n;i++)  
        s=s+v[i]+" ";  
    return s;  
}
```

```
public int size() {  
    return n;  
}
```

```
public Object get(int i) {  
    return v[i];  
}  
}
```

Rezolvare

```
// clasa iterator pentru vectori
class ArrayEnum implements Enumeration {
    private int i;
    private MyArray a;

    ArrayEnum (MyArray a) {
        this.a =a;
        i=0;
    }

    public Object nextElement() {
        return a.get(i++);
    }

    public boolean hasMoreElements() {
        return i < a.size();
    }
}
```

Rezolvare

```
class ArrayDemo {
    // afisare prin enumerare
    public static void main ( String av[]) {
        MyArray a = new MyArray(10);
        for (int i=1;i<11;i++)
            a.add(""+i);
        Enumeration it = a.elements();
        while (it.hasMoreElements())
            System.out.print (it.nextElement()+" ");
        System.out.println();
        System.out.println(a);
    }
}
```