

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.

Tipuri de date reprezentate:

- **vectors**
- **liste înlanțuite**
- **stive**
- **mulțimi matematice**
- **tabele de dispersie, etc.**

Tipul de date al elementelor dintr-o colecție este **Object**.

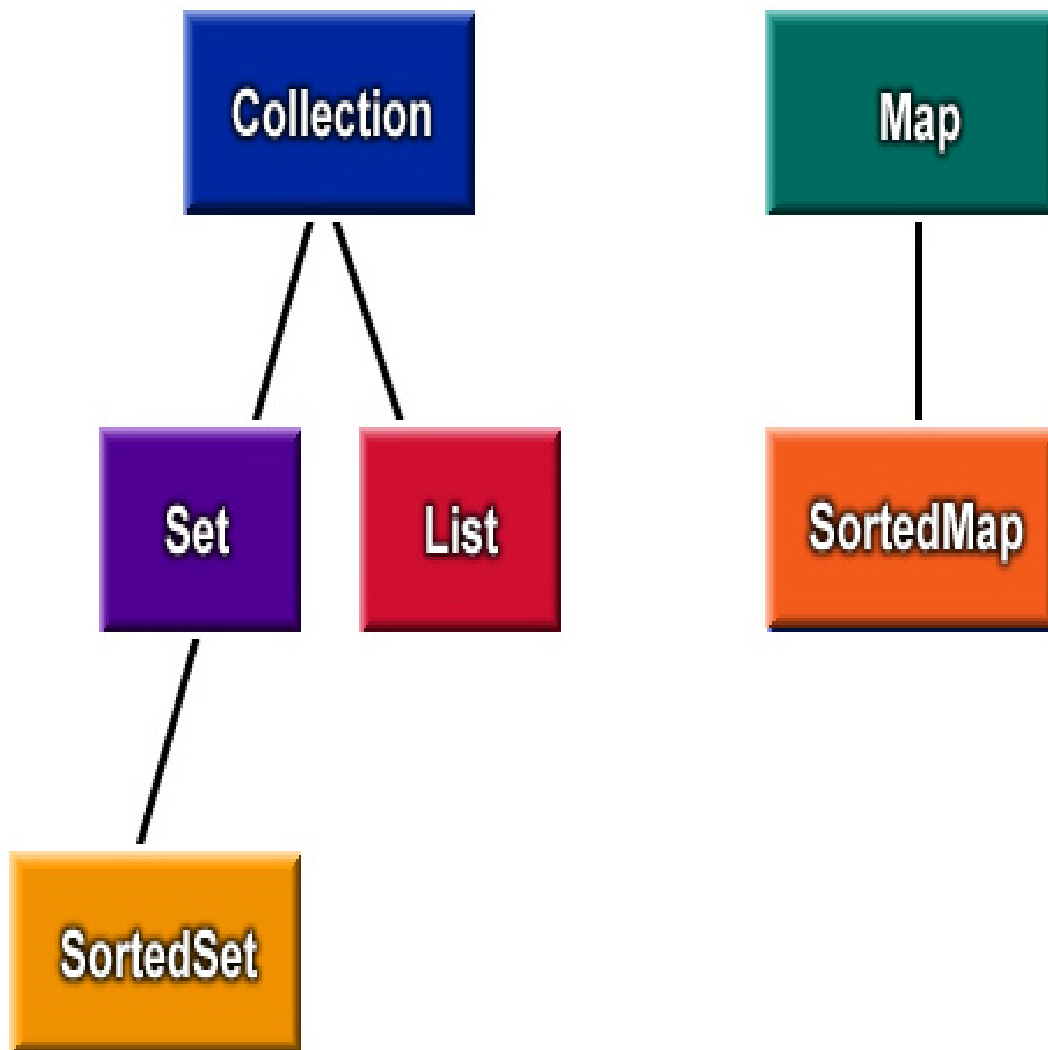
Arhitectura colecțiilor

- **Interfețe:**
tipuri abstracte de date.
- **Implementări:**
tipuri de date reutilizabile.
- **Algoritmi polimorfici:**
funcționalitate reutilizabilă.

Avantaje:

- Reducerea efortului de programare
- Creșterea vitezei și calității programului

Interfețe ce descriu colecții



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();
    Object[] toArray(Object a[]);
}
```

Set

Mulțime în sens matematic.

O mulțime nu poate avea elemente duplicate.

$\nexists o1, o2$ cu proprietatea $o1.equals(o2)$.

Implementări: **HashSet** și **TreeSet**.

SortedSet

Mulțime cu elemente **sortate**.

Ordonarea elementelor este **naturală**, sau dată de un **comparator**.

$\forall o1, o2$, apelul $o1.compareTo(o2)$ (sau $comparator.compare(o1, o2)$) trebuie să fie valid.

```
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**

List

Liste de elemente indexate.

```
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);
}
```

Implementări: **ArrayList**, **LinkedList**,
Vector.

Map

Structuri de tip: **cheie - element**.

```
public interface Map {
    // Metode cu caracter general
    ...
    // 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);
    // Operatii la nivel de multime
    void putAll(Map t);
    // Vizualizari ale colectiei
    public Set keySet();
    public Collection values();
    public Set entrySet();
}
```

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**

Implementări ale colecțiilor

< Implementare > < Interfața >

Interfața	Clasa
Set	HashSet
SortedSet	TreeSet
List	ArrayList, LinkedList Vector
Map	HashMap Hashtable
SortedMap	TreeMap

Organizare ierarhică

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

AbstractMap - HashMap, TreeMap, Hashtable

In vechea ierarhie:

Dictionary - Hashtable - Properties

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 ?

Listing 1: Comparare ArrayList - 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));
    }

    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	LinkedList
add	0.12	0.14
get	0.01	87.45
remove	12.05	0.01

Concluzia: alegerea unei anumite clase depinde de natura problemei ce trebuie rezolvată.

Algoritmi polimorfici

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
- unmodifiable *TipColectie*
- synchronized *TipColectie*

Tipuri generice

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);
```

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

Iteratori și enumerări

Parcurgerea secvențială a unei colecții, indiferent dacă este indexată sau nu.

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

```
// 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));
}
```

Listing 2: 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)

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

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