



Academia Tehnică Militară "Ferdinand I"
Facultatea de Sisteme Informaticе și Securitate Cibernetică

PARADIGME DE PROGRAMARE (ÎN JAVA)

CURS 4 - PROGRAMARE ORIENTATĂ PE OBIECTE (OOP)

COL(R) TRAIAN NICULA

TEMATICĂ DE CURS

- Introducere în paradigme de programare (1)
- **Programare orientată pe obiecte (OOP) (3/3)**
- Programare funcțională și asincronă (3)
- Programare reactivă (1)
- Servicii web REST (2)
- Quarkus Framework (4)

CUPRINS CURS

- Enumerații
- Autoboxing/Unboxing
- Adnotări
- Generice
- Colecții
- Bibliografie



ENUMERAȚII

- Sunt **liste de constante** care definesc un **nou tip de date**
- Un obiect de tip enumerație **poate stoca doar valorile definite de listă**
- Exemple: enumerație cu lunile anului, enumerație cu coduri de stare ale unui proces (succes, așteptare, eroare, reîncercare)
- O enumerație se creează cu cuvântul cheie **enum**:

```
// An enumeration of transportation.
```

```
enum Transport {  
    CAR, TRUCK, AIRPLANE, TRAIN, BOAT  
}
```

- CAR, TRUCK etc. sunt constantele enumerației, membrii publici și statici ai tipului **Transport**
- Chiar dacă definesc un tip **nu se instanțiază cu new**

ENUMERAȚII

```
package atm.paradigms.tests;

public enum Transport {
    CAR, TRUCK, AIRPLANE, TRAIN, BOAT
}
```

Rezultat:

t value: CAR

t contains CAR

Is a CAR

```
package atm.paradigms.tests;

public class TransportTest {
    public static void main(String[] args) {
        // declare a reference and assign
        Transport t = Transport.CAR;
        System.out.println("t value: " + t);
        // compare with if
        if (t == Transport.CAR) {
            System.out.println("t contains CAR");
        }
        // switch with enum
        switch (t) {
            case CAR: // without type
                System.out.println("Is a CAR");
                break;
            case TRUCK:
                System.out.println("Is a TRUCK");
                break;
        }
    }
}
```

ENUMERAȚII

- Enumerațiile au 2 metode predefinite:
- **values()** – Returnează un vector conținând o listă a constantelor

```
public static enum-type[ ] values( )
```

- **valueOf()** – Returnează constanta corespunzătoare sirului de caractere transmis ca parametru

```
public static enum-type valueOf(String str)
```

ENUMERAȚII

```
package atm.paradigms.tests;

public enum Transport {
    CAR, TRUCK, AIRPLANE, TRAIN, BOAT
}
```

Returnează constanta
enumerației corespunzătoare
unei String primit ca parametru

```
package atm.paradigms.tests;

public class TransportTest {
    public static void main(String[] args) {
        // use values()
        Transport allTransports[] = Transport.values();
        for (Transport t : allTransports)
            System.out.println(t);
        // use valueOf()
        Transport tp = Transport.valueOf("AIRPLANE");
        System.out.println("tp contains " + tp);
    }
}
```

Rezultat:

CAR
TRUCK
AIRPLANE
TRAIN
BOAT
tp contains AIRPLANE



ENUMERAȚII

- Enumerațiile pot defini **constructori, metode și variabile de instanță**
- Dacă se definește constructor acesta este chemat la crearea constantelor
- Enumerațiile **nu pot moșteni și nu pot fi extinse**

```
package atm.paradigms.tests;

public enum Transport {
    CAR(65), TRUCK(55), AIRPLANE(600),
    TRAIN(70), BOAT(22);

    private int speed;

    Transport(int speed) {
        this.speed = speed;
    }

    public int getSpeed() {
        return speed;
    }
}
```

```
package atm.paradigms.tests;

public class TransportTest {
    public static void main(String[] args) {
        // print speed for each vehicle
        Transport allTransports[] = Transport.values();
        for (Transport t : allTransports)
            System.out.println(t.getSpeed());
    }
}
```

Rezultat:

65

55

600

70

22

AUTOBOXING/UNBOXING

- Uneori este nevoie ca **tipurile primitive să fie reprezentate ca obiecte**
- **Simplifică conversia tipurilor primitive în obiecte și invers**
- În acest scop există **clase speciale** în Java numite **type wrappers**
- Clasele de tip **type wrappers sunt Double, Float, Long, Integer, Short, Byte, Character, și Boolean**
- **Procesul de încapsulare** a unei valori într-un obiect se numește **boxing** iar procesul invers **unboxing**
- Până la Java 5 procesul se făcea manual



AUTOBOXING/UNBOXING CONVERSIE MANUALĂ

```
package atm.paradigms.tests;

public class Wrap {
    public static void main(String[] args) {
        try{
            // manually boxing
            Integer iObj = Integer.parseInt("100");
            System.out.println("Integer: " + iObj);
            // manually unboxing
            int i = iObj.intValue();
            System.out.println("int value: " + i);
        }
        catch (NumberFormatException e){
            e.printStackTrace();
        }
    }
}
```

Metoda `parseInt()` a clasei `Integer` poate genera excepția `NumberFormatException`

Rezultat:
`Integer: 100`
`int value: 100`



AUTOBOXING/UNBOXING

- Autoboxing este procesul prin care tipurile primitive sunt încapsulate automat în obiecte aparținând claselor **type wrappers**
- Obiectul se construiește prin atribuirea unei referințe **type wrappers** o valoare primitivă

```
Integer iOb = 100; // autobox an int
```

- Operația inversă se face prin atribuirea unui obiect unei variabile de tip primitiv

```
int i = iOb; // auto-unbox
```

```
package atm.paradigms.tests;

public class Wrap {
    public static void main(String[] args) {
        Integer iObj = 100; // boxing
        int i = iObj; // unboxing
        iObj++; // work with expressions
        System.out.println(i + " " + iObj);
    }
}
```

Rezultat:
100 101



ADNOTĂRI

- Facilitate introdusă în Java 5 care permite **încapsularea de informații suplimentare** în fișierul sursă
- Deși nu schimbă logica programului, **adnotările** sunt folosite de diferite unelte pe timpul dezvoltării programului, la instalarea (deployment) acestuia și la rulare
- Exemplu de adnotare:

```
// A simple annotation type.  
  
@interface MyAnno {  
  
    String str();  
  
    int val();  
}
```

- **@interface** anunță compilatorul că a fost declarată o adnotare
- Adnotările **constau doar din metode** care se comportă ca și câmpuri de date



ADNOTĂRI

- Tipurile adnotări extind interfața **Annotation** aflată în pachetul `java.lang.annotation`
- Se pot adăuga adnotări **claselor, metodelor, variabilelor de instanță, parametrilor și enumerațiilor**
- Adnotarea se aplică unei metode ca în exemplul de mai jos

```
// Annotate a method.  
  
@MyAnno(str = "Annotation Example", val = 100)  
public static void myMeth() { // ...}
```

- Adnotarea este precedată de `@` iar în parantezele rotunde se initializează membrii acesteia
- Adnotările fără parametrii se numesc adnotări de marcaj
- Adnotări incluse în Java: `@Override`, `@Deprecated`, și `@SuppressWarnings`



GENERICE

- Înseamnă **tipuri parametrizate**
- **Tipurile parametrizate** sunt importante deoarece **permisă crearea de clase, interfețe și metode** în care **tipul de date** pe care operează este **specificat ca parametru**
- Codul generic operează pe tipuri de date transmise ca parametrii
- Se pot crea **algoritmi independenți de tipul de date** care apoi se pot aplica diferitelor tipuri
- Soluția anterioară introducerii genericelor se baza pe faptul că **tipul Object** este superclasa pentru toate subclasele din Java
- Problema este că **nu se poate face întotdeauna** în siguranță **conversia de tip**



GENERICE EXEMPLU

Clasă generică ce folosește intern tipul parametrizat T

```
package atm.paradigms.tests;

public class Generic<T> {
    private T obj;
    public Generic(T o){
        this.obj = o;
    }
    public T getObj() {
        return obj;
    }
    public String getType(){
        return this.obj.getClass().getName();
    }
}
```

```
package atm.paradigms.tests;

public class GenericTest {
    public static void main(String[] args) {
        Generic<Integer> g = new Generic<Integer>(100);
        System.out.println(g.getType() + ": "
                           + g.getObj());
        Generic<String> g1 = new Generic<String>("test");
        System.out.println(g1.getType() + ": "
                           + g1.getObj());
    }
}
```

Rezultat:
`java.lang.Integer: 100`
`java.lang.String: test`



GENERICE

- Clasa generică este declarată cu

```
class Gen<T> { //...
```

- T este numele parametrului de tip ce va fi înlocuit cu tipul transmis clasei generice la crearea obiectului
- Parametrul poate fi orice literă dar se recomandă să fie un singur caracter cu literă mare
- T este folosit pentru declararea unui obiect și este un înlocuitor pentru tipul ce va fi specificat când obiectul de tip generic este creat
- În generice se pot trece ca parametrii doar tipuri clasă. Nu sunt permise tipurile primitive dar pot fi folosite type wrappers



GENERICE

Clasă generică având 2 tipuri parametrizate

```
package atm.paradigms.tests;

public class Generic<T, V> {
    private T o1;
    private V o2;
    public Generic(T o1, V o2){
        this.o1 = o1;
        this.o2 = o2;
    }
    public T getTObj() {
        return o1;
    }
    public V getVObj() {
        return o2;
    }
    public String getTypes(){
        return "T: "
            + this.o1.getClass().getName()
            + "\nV: "
            + this.o2.getClass().getName();
    }
}
```

```
package atm.paradigms.tests;

public class GenericTest {
    public static void main(String[] args) {
        Generic<Integer, String> obj =
            new Generic<Integer, String>(100, "Paradigms");
        System.out.println(obj.getTypes());
        System.out.println(obj.getTObj());
        System.out.println(obj.getVObj());
    }
}
```

Rezultat:
 T: java.lang.Integer
 V: java.lang.String
 100
 Paradigms

GENERICE RESTRÂNGEREA TIPURILOR

- În exemplele anterioare parametrii de tip puteau fi orice clasă
- Uneori este necesar **să se limiteze tipurile de clase permise** ca parametrii de tip
- Java permite **declararea superclasei** din care parametrii de tip pot fi derivați

<T extends superclass>

- **T poate fi înlocuit de superclasă și subclasele acesteia stabilind o limită superioară de tip**
- **Number** este superclasă pentru toate tipurile numerice **Byte, Short, Integer, Long, Float, și Double**



GENERICE RESTRÂNGEREA TIPURILOR

```
package atm.paradigms.tests;

public class NumberFunc<T extends Number>{
    private T value;
    public NumberFunc(T value){
        this.value = value;
    }
    // reciprocal
    public double reciprocal(){
        return 1 / value.doubleValue();
    }
    // fractional part
    public double fractional(){
        return value.doubleValue()
            - value.intValue();
    }
}
```

Tipul parametrizat este restrâns la **Number** și subclasele acestuia.
Se utilizează metodele clasei **Number**

```
package atm.paradigms.tests;

public class NumberFuncTest {
    public static void main(String[] args) {
        NumberFunc<Integer> i = new NumberFunc<Integer>(5);
        System.out.println("Reciprocal: " + i.reciprocal());
        NumberFunc<Double> d = new NumberFunc<Double>(6.88);
        System.out.println("Fractional part: "
            + d.fractional());
    }
}
```

Rezultat:
Reciprocal: 0.2
Fractional part: 0.8799999999999999

GENERICE

TIPURI GENERICE CA PARAMETRII PENTRU METODE

- **Stabilirea limitei superioare pentru tipurile acceptate ca parametrii de o metodă se face cu sintaza**

`<? extends superclass>`

- **superclass** - doar **clasa și subclasele acesteia sunt acceptați ca parametru**
- Exemplu:

```
// Here, the ? will match A or any class type  
// that extends A.
```

```
static void test(Gen<? extends A> o) {  
    // ...  
}
```

- Este posibilă **limitarea inferioară a tipurilor permise ca parametrii în metodă**

`<? super subclass>`

GENERICE

METODE GENERICE

- Pot fi declarate **metode generice care folosesc proprii parametrii de tip**
- Metodele generice pot fi incluse în clase negenerice
- Sintaxa pentru o metodă generică

<type-param-list> ret-type meth-name(param-list) { // ...

- **type-param-list este o listă de parametri de tip separați cu virgulă care preced tipul returnat de metodă**



GENERICE

METODE GENERICE

```
package atm.paradigms.tests;

public class GenericMeth {
    static <T, V extends T> boolean testEqual(T[] x, V[] y){
        // same length
        if (x.length != y.length) return false;
        // same values
        for (int i = 0; i < x.length; i++){
            if (x[i] != y[i]) return false;
        }
        return true;
    }
    public static void main(String[] args) {
        Number nums1[] = {1, 2, 3, 4, 5};
        Byte nums2[] = {1, 2, 3, 4, 5};
        if (testEqual(nums1, nums2))
            System.out.println("Equal array");
        else
            System.out.println("Different array\nWhy?");
    }
}
```

Metodă generică într-o clasă negenerică. Se definesc tipurile acceptate înainte de tipul returnat. Aici V este de tip T sau o subclasă.

Rezultat:
Different array
Why?

GENERICE CONSTRUCTORI GENERICI

- Putem avea **constructori generici** în clase negenerice

```
package atm.paradigms.tests;

public class Sumation {
    private int sum;
    public <T extends Number> Sumation(T val) {
        this.sum = 0;
        for (int i = 0; i <= val.intValue(); i++)
            sum += i;
    }
    public int getSum() {
        return sum;
    }
}
```

```
package atm.paradigms.tests;

public class SumationTest {
    public static void main(String[] args) {
        Sumation o = new Sumation(10.5);
        System.out.println("Sumation is "
                           + o.getSum());
    }
}
```

GENERICE INTERFEȚE

- **Interfețele generice sunt specificate în același mod cu clasele generice**
- Forma generală

interface *interface-name*<*type-param-list*> { // ...

- ***type-param-list*** este listă de parametrii de tip separați cu virgulă
- Sintaxa pentru **implementarea unei interfețe generice**

class *class-name*<*type-param-list*>

implements *interface-name*<*type-param-list*> {



GENERICE INTERFEȚE

```
package atm.paradigms.tests;

public interface Containment<T> {
    // The contains() method tests if a
    // specific item is contained within
    // an object that implements Containment.
    boolean contains(T o);
}
```

```
package atm.paradigms.tests;

public class MyClassTest {
    public static void main(String[] args) {
        Integer x[] = {1, 2, 3, 4, 5};
        MyClass<Integer> obj =
            new MyClass<Integer>(x);
        if (obj.contains(2)){
            System.out.println("2 was found");
        }
        // if (obj.contains(2.5)) Illegal! Why?
    }
}
```

```
package atm.paradigms.tests;

public class MyClass<T> implements Containment<T> {
    private T[] array;
    public MyClass(T[] o){
        this.array = o;
    }
    @Override
    public boolean contains(T o) {
        for (T t : array)
            if (t.equals(o)) return true;
        return false;
    }
}
```

Rezultat:
2 was found

Clasa generică implementează interfața generică cu metoda **contains**.
Parametrul T poate fi doar referință de tip clasă și nu tip primitiv.

COLECȚII

- Prin **colecții** (Collections Framework) Java a **standardizat modul în care sunt manipulate grupurile de obiecte**
- Până la introducerea colecțiilor existau 4 clase pentru manipularea grupurilor de obiecte: **Dictionary, Vector, Stack și Properties**
- **Colecții** au fost create pentru:
 - Înaltă performanță
 - Interoperabilitate între ele
 - Ușurință în adoptare și extindere
- **Colecții** sunt construite pe un **set standard de interfețe**
- **Algoritmii** care operează pe colecții sunt definiți ca metode statice în clasa **Collections**
- Interfața **Iterator** oferă o **modalitate standardizată și generală de accesare a elementelor** colecției unul câte unul
- Chiar dacă nu sunt colecții, **framework-ul definește și interfețe de tip Maps** pentru stocarea de perechi cheie/valoare



COLECȚII INTERFEȚE

Interfață	Descriere
Collection	Se află în vârful ierarhiei de interfețe pentru colecții. Permite lucrul cu grupuri de obiecte
List	Extinde Collection pentru manipularea listelor de obiecte
Queue	Extinde Collection pentru manipularea listelor speciale în care obiectele sunt adăugate și șterse doar de la un capăt
Set	Extinde Collection pentru manipularea seturilor, adică liste care conțin obiecte unice
SortedSet	Extinde Set pentru manipularea seturilor sortate

Alte interfețe utile sunt **Comparator, RandomAccess, Iterator, ListIterator și Spliterator**



COLECTII

INTERFAȚA COLLECTION

- Collection este o interfață generică

interface Collection<E>

- E specifică tipul de obiecte stocate de colecție
- Câteva metode utile:

Metodă	Descriere
boolean add(E obj) boolean remove(E obj)	Adaugă/șterge obiect la colecție
boolean addAll(Collection<? extends E> c) boolean removeAll(Collection<? extends E> c)	Adaugă/șterge toate elementele lui c la colecție
void clear()	Șterge toate elementele din colecție
boolean contains(Object obj)	Caută un element în colecție
boolean containsAll(Collection<?> c)	Caută toate elementele colecției c în colecția curentă
boolean isEmpty()	Testează dacă colecția conține elemente
Iterator<E> iterator()	Returnează un Iterator pentru colecție
int size()	Returnează numărul de elemente din colecție



COLECȚII

INTERFAȚA LIST

- Interfața List extinde Collection și declară comportarea unei colecții care stochează o secvență de elemente
- List este o interfață generică în care pot fi inserate și accesate elemente prin poziția lor în listă folosind un indice ce pleacă de la 0
- Metode specifice:

Metodă	Descriere
boolean add(int index, E obj) boolean remove(int index)	Adaugă/șterge obiect la colecție la/de la index. Obiectele existente sunt deplasate
boolean addAll(int index, Collection<? extends E> c)	Adaugă/șterge toate elementele lui c la colecție începând cu poziția index
E get((int index)	Returnează obiectul stocat la poziția index
E set((int index, E obj)	Setează cu obj poziția index din colecție
int indexOf(Object obj)	Returnează indexul primei instanțe a lui obj din listă
ListIterator<E> listIterator()	Returnează un Iterator pentru listă de la indexul 0
ListIterator<E> listIterator(int index)	Returnează un Iterator pentru listă de la index



COLECȚII

CLASA ARRAYLIST

- **ArrayList** extinde **AbstratList** și implementează interfața **List**
- **ArrayList** este o clasă generică

`class ArrayList<E>`

- **Suport pentru vectori dinamici** care pot crește sau scădea în dimensiune
- Constructori:

`ArrayList()`

`ArrayList(Collection<? extends E> c)`

`ArrayList(int capacity)`

- Primul construiește o listă goală, al doilea construiește o listă inițializată cu elementele din colecție, iar la treilea construiește o listă de capacitate specificată

COLECȚII

CLASA ARRAYLIST

```
ackage atm.paradigms.tests;

import java.util.ArrayList;
import java.util.List;

public class ArrayListTest {
    public static void main(String[] args) {
        List<String> al = new ArrayList<String>();
        al.add("A");
        al.add("B");
        al.add("C");
        al.add("D");
        System.out.println("List size: " + al.size());
        al.remove("B");
        al.remove(2);
        System.out.println("List: " + al);
    }
}
```

Rezultat:
*List size: 4
List: [A, C]*



COLECȚII

CLASA ARRAYLIST

- Pentru a obține un vector din listă se folosesc metodele

object[] toArray()

<T> T[] toArray(T array[])

- Prima returnează un vector de tip Object, iar a doua tipul corespunzător

```
package atm.paradigms.tests;

import java.util.ArrayList;
import java.util.List;

public class ArrayListTest {
    public static void main(String[] args) {
        List<Integer> al = new ArrayList<Integer>();
        al.add(1);
        al.add(2);
        al.add(3);
        al.add(4);
        Integer ia[] = new Integer[al.size()];
        ia = al.toArray(ia);
        System.out.println("Array size: " + ia.length);
    }
}
```

Rezultat:
Array size: 4

COLECȚII

INTERFAȚA SET

- **Interfața Set extinde *Collection* și definește o colecție care nu permite duplicarea elementelor**
- **Set returnează *false* dacă se încearcă adăugarea unui element duplicat**
- Set este o interfață generică
 - **interface Set<E>**
- E specifică tipul de obiecte pe care le ține colecția Set



COLECȚII

CLASA HASHSET

- Extinde *AbstractSet* și implementează interfața *Set*
- Folosește folosește mecanismul de *hashing* pentru stocare
- **Conținutul informațional cu valoare unică sau cheia este convertită la hash code**
- *hash code* este folosit apoi ca index la care datele asociate cu cheia sunt stocate
- Mecanismul de *hashing* permite ca **timpul de execuție** al metodelor *add()*, *contains()*, *remove()* și *size()* să rămână **constant** chiar și pentru seturi mari
- **HashSet nu garantează ordinea elementelor**, se folosesc seturi sortate precum **TreeSet**

- Constructori

HashSet()

HashSet(Collection<? extends E> c)

HashSet(int capacity)

HashSet(int capacity, float fillRatio)

COLECȚII

CLASA HASHSET

```
package atm.paradigms.tests;

import java.util.HashSet;
import java.util.Set;

public class HashSetTest {
    public static void main(String[] args) {
        Set<String> hs = new HashSet<String>();
        String a = "A";
        // String hashcode
        System.out.println(a.hashCode());
        hs.add(a);
        hs.add("B");
        hs.add("B");
        hs.add("C");
        System.out.println(hs);
        hs.remove(a);
        System.out.println(hs);
    }
}
```

Rezultat:

65

[A, B, C]

[B, C]



COLECȚII

INTERFAȚA SORTEDSET

- Extinde **Set** și descrie **comportamentul unui set sortat în ordine ascendentă**
- Definiția interfeței generice
- interface SortedSet<E>**
- E specifică tipul de obiecte din set
- Metode specifice

Metodă	Descriere
E first()	Returnează primul obiect din set
E last()	Returnează ultimul obiect din set
SortedSet<E> subSet(E start, E end)	Returnează un subset al setului original ale căruia elemente se află între start și end-1
SortedSet<E> tailSet(E start)	Returnează un subset care începe cu start și se termină la capătul setului original



COLECȚII INTERFAȚA NAVIGABLESET

- Extinde **SortedSet** și descrie comportamentul unui set ce permite găsirea elementelor cele mai apropiate de o valoare dată
- Metode specifice:

Metodă	Descriere
E ceiling(E obj)	Caută în set cel mai mic element e care respectă condiția $e \geq \text{obj}$
E floor(E obj)	Caută în set cel mai mic element e care respectă condiția $e \leq \text{obj}$
E higher(E obj)	Caută în set cel mai mic element e care respectă condiția $e > \text{obj}$
E lower(E obj)	Caută în set cel mai mic element e care respectă condiția $e < \text{obj}$
E pollFirst()	Returnează și șterge primul element din set
E pollLast()	Returnează și șterge ultimul element din set



COLECȚII

CLASA TREESSET

- Extinde **AbstractSet** și implementează interfața **NavigableSet**
- Clasa **creează o colecție bazată pe un arbore** care stochează obiecte stocate în ordine ascendentă
- Accesul și recuperarea datelor se face foarte rapid
- Clasa **TreeSet** este o alegere perfectă pentru stocarea de cantități mari de informație care trebuie găsită rapid
 - TreeSet este o clasă generică de forma:
`class TreeSet<E>`
 - E specifică tipului de obiecte din set

COLECȚII

CLASA TREESSET

```
package atm.paradigms.tests;
import java.util.TreeSet;

public class TreeSetTest {
    public static void main(String[] args) {
        TreeSet<String> ts = new TreeSet<String>();
        ts.add("C");
        ts.add("B");
        ts.add("A");
        ts.add("E");
        ts.add("F");
        ts.add("D");
        // automatically sorted objects
        System.out.println(ts);
        System.out.println(ts.first());
        System.out.println(ts.ceiling("B"));
        System.out.println(ts.higher("B"));
        ts.pollFirst();
        ts.pollLast();
        System.out.println(ts);
    }
}
```

Rezultat:

[A, B, C, D, E, F]

A

B

C

[B, C, D, E]



COLECȚII

INTERFAȚA ITERATOR

- Folosită pentru a **parurge elementele unei colecții** pentru a obține sau șterge elemente
- **ListIterator** extinde **Iterator** și permite **traversarea bidirectională a unei liste** și modificarea elementelor
- **Iterator** și **ListIterator** sunt interfețe generice

interface Iterator<E>

interface ListIterator<E>

- Metode specifice **Iterator**

Metodă	Descriere
boolean hasNext()	Returnează true dacă mai sunt elemente în colecție
E next()	Returnează următorul element din colecție
void remove()	Șterge elementul curent



COLECȚII

INTERFAȚA LISTITERATOR

- Metode specifice:

Metodă	Descriere
boolean hasPrevious()	Returnează true dacă există element anterior în listă
E nextIndex()	Returnează indicele următorului element din listă. Dacă nu există returnează dimensiunea listei
E previous()	Returnează elementul anterior
E previousIndex()	Returnează indicele elementului anterior din listă. Dacă nu există returnează -1
void add(E obj)	Inserează obj în listă în fața elementului care va fi returnat prin invocarea metodei next()
void set(E obj)	Atribuie obj elementului curent. Acesta este elementul returnat prin ultima invocare a metodelor next() sau previous()

COLECȚII

FOLOSIREA INTERFETEI ITERATOR

- Pentru parcurgerea colecției folosind un **Iterator** se urmează pașii:
 - **Se invocă metoda *iterator()*** pe clasa care implementează colecția. Iteratorul se află la începutul acesteia
 - **Se creează o buclă** prin invocarea metodei ***hasNext()***
 - În buclă **se obțin elementele** unul câte unul prin invocarea ***next()***
- Pentru colecțiile care implementează **List** se poate obține un **Iterator** cu metoda ***listIterator()*** care permite accesul la listă în ambele direcții

COLECȚII

FOLOSIREA INTERFETEI ITERATOR

```
package atm.paradigms.tests;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.ListIterator;

public class IteratorTest {
    public static void main(String[] args) {
        List<String> al =
            new ArrayList<String>();
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("F");
        al.add("D");
    }
}
```

```
Iterator<String> it = al.iterator();
while(it.hasNext()){
    String e = it.next();
    System.out.print(e + " ");
}
System.out.println();
ListIterator<String> lit = al.listIterator();
while(lit.hasNext()){
    String e = lit.next();
    if (e.equals("B"))
        lit.add("W");
}
System.out.println(al);
}
```

Rezultat:

C A E B F D
[C, A, E, B, W, F, D]



COLECȚII INTERFAȚA QUEUE

- Extinde interfața **Collection** și declară comportarea unei cozi (first-in, first-out)

Metodă	Descriere
E element()	Returnează elementul din capătul cozii fără al șterge. Generează excepție dacă coada este goală
boolean offer(E obj)	Încearcă să adauge obj la coadă
E peek()	Returnează elementul din capătul cozii fără al șterge. Returnează null dacă coada este goală
E poll()	Returnează și șterge elementul din capătul cozii. Returnează null dacă coada este goală
E remove()	Șterge și returnează elementul din capătul cozii. Generează excepție dacă coada este goală

COLECȚII

INTERFAȚA DEQUE

- Interfața **Deque** (coada dublă) descrie comportarea pentru o **coadă cu două capete** (double-ended queue)
- **Deque** poate funcționa ca o coadă standard (first-in, first-out) sau ca stivă (last-in, first-out)
- Metodele moștenite de la **Collection** și **Queue** au variante pentru cele 2 capete ale cozii **addFirst(E obj)**, **addLast(E obj)**, **E getFirst()**, **E getLast()**, **boolean offerFirst(E obj)**, **boolean offerLast(E obj)** etc.

Metodă	Descriere
E pop()	Returnează elementul de la capătul de început cozii și îl șterge. Generează excepție dacă coada este goală
void push(E obj)	Adaugă obj la capătul de început cozii.
Iterator<E> descendingIterator()	Returnează un iterator care parcurge coada de la sfârșit spre începutul acesteia

COLECȚII CLASA LINKEDLIST

- Extinde *AbstractSequentialList* și implementează interfețele *List*, *Queue* și *Deque*
- Implementează o structură de date de tip listă înlănțuită
- Declarație și constructori:

```
class LinkedList<E>
```

```
LinkedList()
```

```
LinkedList(Collection<? extends E> c)
```

COLECȚII

CLASA LINKEDLIST

```
package atm.paradigms.tests;
import java.util.LinkedList;

public class LinkedListTest {
    public static void main(String[] args) {
        LinkedList<String> ll = new LinkedList<String>();
        ll.add("F");
        ll.add("B");
        ll.add("D");
        ll.add("C");
        ll.addFirst("A");
        ll.addLast("Z");
        System.out.println(ll);
        ll.add(1, "E");
        System.out.println(ll);
        ll.remove("F");
        ll.removeFirst();
        ll.removeLast();
        System.out.println(ll);
    }
}
```

Rezultat:

[A, F, B, D, C, Z]

[A, E, F, B, D, C, Z]

[E, B, D, C]



COLECȚII

CLASA LINKEDLIST

```
package atm.paradigms.tests;

public class Address {
    private String name;
    private String street;
    private String city;
    private String state;
    public Address(String name, String street,
                  String city, String state) {
        this.name = name;
        this.street = street;
        this.city = city;
        this.state = state;
    }
    @Override
    public String toString() {
        return name + "\n" + street + "\n"
               + city + " " + state;
    }
}
```

```
package atm.paradigms.tests;

import java.util.LinkedList;

public class AddressTest {
    public static void main(String[] args) {
        LinkedList<Address> at = new
        LinkedList<Address>();
        at.add(new Address("Ion Popescu",
                           "Calea Rahovei", "S.5", "Bucuresti"));
        at.add(new Address("Costin Georgescu",
                           "Sos. Alexandriei", "S.5", "Bucuresti"));
        at.add(new Address("Vasile Mirea",
                           "Str. Teius", "S.5", "Bucuresti"));
        for (Address a : at){
            System.out.println(a + "\n");
        }
    }
}
```



COLECȚII INTERFAȚA MAP

- **Map** este un obiect care stochează perechea cheie/valoare
- Atât **cheile** cât și **valorile** sunt obiecte
- **Cheile trebuie să fie unice**, dar **valorile pot fi duplicate**
- Pentru un **Map** nu se poate obține un **Iterator** și nu se poate parcurge direct cu bucla **for**
- Interfețe suportate de **Map**

Interfață	Descriere
Map	Asociază chei unice la valori
Map.Entry	Descrie un element cheie/valoare
SortedMap	Extinde Map pentru a menține cheile în ordine ascendentă
NavigableMap	Extinde SortedMap pentru a căuta și returna perechi cheie/valoare folosind criteriul celei mai apropiate potriviri



COLECȚII INTERFAȚA MAP

- Declarația interfeței **Map**

interface Map<K, V>

- K specifică tipul pentru chei iar V tipul pentru valori
- Metode specifice:

Metodă	Descriere
V get(Object k)	Returnează valoarea asociată cu cheia k. Dacă nu găsește cheia returnează null
V put(K k, V v)	Pune un element cheie/valoare în Map. Îl suprascrie dacă cheie există
Set<Map.Entry<K, V>> entrySet()	Returnează un set care conține elementele cheie/valoare din Map
Set<K> keyset()	Returnează un set care conține cheile din Map
V remove(Object k)	Șterge elementul cheie/valoare cu cheia egală cu k
Collection<V> values()	Returnează o colecție conținând valorile din Map

COLECȚII

INTERFAȚA MAP.ENTRY

- Interfața **Map.Entry** permite manipularea perechilor cheie/valoare
- Metode specifice utile:

Metodă	Descriere
K getKey()	Returnează cheia dintr-o pereche cheie/valoare
V getValue()	Returnează valoarea dintr-o pereche cheie/valoare
V setValue(V v)	Setează valoarea pentru o pereche cheie/valoare



COLECȚII

CLASA HASHMAP

- Extinde clasa **AbstractMap** și implementează interfața **Map**
- Folosește un tablou de **hash** pentru stocare **Map**
- Este o clasă generică:

class HashMap<K, V>

- K specifică tipul pentru chei iar V tipul pentru valori
- Constructori

HashMap()

HashMap(Map<? extends K, ? extends V> m)

HashMap(int capacity)

HashMap(int capacity, float fillRatio)

- Ordinea elementelor nu este garantată asemănător **HashSet**



COLECȚII CLASA HASHMAP

```
package atm.paradigms.tests;
import java.util.*;
public class HashMapTest {
    public static void main(String[] args) {
        Map<String, Double> hm = new HashMap<String, Double>();
        hm.put("D", 5.5);
        hm.put("B", 4.4);
        hm.put("C", 2.2);
        hm.put("A", 1.1);
        // set from map
        Set<Map.Entry<String, Double>> set = hm.entrySet();
        for(Map.Entry<String, Double> me : set){
            System.out.print(me.getKey() + ": ");
            System.out.println(me.getValue());
        }
        double value = hm.get("A");
        hm.put("A", value + 100);
        System.out.println(hm);
    }
}
```

Rezultat:

A: 1.1

B: 4.4

C: 2.2

D: 5.5

{A=101.1, B=4.4, C=2.2, D=5.5}

COLECȚII COMPARATORS

- Anumite clase precum **TreeSet** sau **TreeMap** stochează obiectele sortate în ordine
- Pentru a controla modul în care obiectele sunt sortate Java oferă interfața **Comparator**

interface Comparator<T>

- T specifică tipul de obiecte care se compară
- Până la Java 8 interfața comparator a oferit metodele **compare()** și **equals()**
- Metoda **compare()**

int compare(T obj1, T obj2)

- obj1 și obj2 sunt obiectele care se compară, iar metoda **returnează 0 dacă obiecte sunt egale, valoare pozitivă dacă obj1 este mai mare decât obj2 și valoare negativă dacă obj1 este mai mic decât obj2**

COLECȚII COMPARATORS

- Metoda **equals()**
boolean equals(Object obj)
- **obj** este obiectul cu care se testează egalitatea
- Metode introduse în Java 8:
- **reversed()** care returnează un Comparator cu ordine inversată față de cel inițial
- **thenComparing()** returnează un Comparator care efectuează o a 2-a comparație atunci când prima comparație returnează egal

COLECȚII COMPARATORS

```
package atm.paradigms.tests;  
import java.util.Comparator;  
  
public class MyComp  
    implements Comparator<String> {  
  
    @Override  
    public int compare(String o1, String o2) {  
        // reverse the comparison  
        return o2.compareTo(o1);  
    }  
}
```

Clasa MyComp implementează interfața Comparator și inversează ordinea de sortare comparând o2 cu o1.

```
package atm.paradigms.tests;  
  
import java.util.Set;  
import java.util.TreeSet;  
  
public class CompTest {  
    public static void main(String[] args) {  
        Set<String> ts =  
            new TreeSet<String>(new MyComp());  
        ts.add("C");  
        ts.add("B");  
        ts.add("A");  
        ts.add("E");  
        ts.add("D");  
        for(String el : ts)  
            System.out.print(el + " ");  
        System.out.println();  
    }  
}
```

Folosește comparatorul creat.

Rezultat:
E D C B A

COLECȚII COMPARATORS

- Implicit Java folosește ordinea naturală de stocare (A înainte de B, 1 înainte de 2 etc.)
- În exemplul anterior putem utiliza un Comparator cu ordine naturală pe care aplicăm metoda **reversed()**

```
package atm.paradigms.tests;
import java.util.Comparator;

public class MyComp
    implements Comparator<String> {

    @Override
    public int compare(String o1, String o2) {
        // natural order sort
        return o1.compareTo(o2);
    }
}
```

Clasa Comparator sortează în ordine naturală. O folosim pentru a aplica reversed() pe ea.

```
package atm.paradigms.tests;

import java.util.Set;
import java.util.TreeSet;

public class CompTest {
    public static void main(String[] args) {
        MyComp mc = new MyComp();
        Set<String> ts =
            new TreeSet<String>(mc.reversed());
        ...
    }
}
```



COLECȚII ALGORITMI

- Colectiile pun la dispozitie algoritmi care pot fi aplicati pe acestea
- Algoritmii sunt implementati ca **metode statice din clasa Collections**
- Metode utile:

Metodă	Descriere
max(Collection) min(Collection)	Returnează elementul maxim sau minim folosind metoda de comparație naturală a obiectelor din colecție
boolean addAll(Collection<? super T> c, T... elements)	Inserează elemente în colecția c
int binarySearch(List<? Extends T> list, T value, Comparator<? super T> c)	Caută <i>value</i> în <i>list</i> ordonată conform <i>c</i> . Returnează poziția valorii în listă sau o valoare negativă dacă nu a găsit nimic
void reverse(List<T> list)	Inversează lista <i>list</i>
sort(List<T> list) sort(List<T> list, Comparator<? super T> c)	Sortează lista folosind ordinea naturală sau un comparator

COLECTII ALGORITMI

```
package atm.paradigms.tests;  
import java.util.*;  
  
public class Algorithms {  
    public static void main(String[] args) {  
        List<Integer> ll = new LinkedList<Integer>();  
        ll.add(-8);  
        ll.add(-10);  
        ll.add(0);  
        ll.add(10);  
        ll.add(8);  
        Comparator<Integer> r = Collections.reverseOrder();  
        Collections.sort(ll, r);  
        for(int i : ll)  
            System.out.print(i + " ");  
        System.out.println();  
        System.out.println("min: " + Collections.min(ll) +  
                           " max: " + Collections.max(ll));  
    }  
}
```

Rezultat:
10 8 0 -8 -10
min: -10 max: 10

Returnează un comparator pe care
îl folosim pentru sortarea listei.



COLECȚII

CLASA ARRAYS

- Clasa *Arrays* pune la dispoziție **metode pentru lucrul cu vectori**
- Metode utile:

Metodă	Descriere
<code>List<T> asList(T... array)</code>	Returnează o listă pe baza unui vector primit ca argument
<code>int binarySearch(T[] array, T value, Comparator<? super T> c)</code>	Caută o valoare într-un vector. Dacă o găsește returnează poziția valorii, dacă nu returnează o valoare negativă
<code>T[] copyOf(U[] source, int len, Class<? extends T[]> resultT)</code>	Returnează o copie a sursei source cu lungimea len și cu tipul specificat de result T. Dacă copia este mai scurtă decât originalul se trunchiază restul elementelor. Dacă len este mai mare decât sursa, se adaugă 0 pentru vectori numerici, obiecte nule pentru vectori de obiecte sau false pentru vectorii de tip boolean
<code>boolean equals(T[] array1, U[] array2)</code>	Returnează true dacă cei 2 vectori sunt echivalenți
<code>void sort(T array[], Comparator<? Super T> c)</code>	Sortează un vector în ordine ascendentă
<code>void fill(Object array[], Object value)</code>	Atribuie o valoare value pentru fiecare element din vector



COLECȚII

CLASA ARRAYS

[Cuprins](#)

```
package atm.paradigms.tests;
import java.util.Arrays;

public class ArraysTest {
    public static void main(String[] args) {
        int array[] = new int[10];
        for( int i = 0; i < 10; i++)
            array[i] = -3 * i;
        print(array);
        Arrays.sort(array);
        print(array);
        int idx = Arrays.binarySearch(array, -9);
        if (idx > 0)
            System.out.println("Found at " + idx);
        else
            System.out.println("Not found");
    }
    static void print(int array[]){
        for (int i : array)
            System.out.print(i + " ");
        System.out.println();
    }
}
```

Rezultat:

0 -3 -6 -9 -12 -15 -18 -21 -24 -27
-27 -24 -21 -18 -15 -12 -9 -6 -3 0

Found at 6

BIBLIOGRAFIE

- **Java™: A Beginner's Guide, Third Edition**, Herbert Schildt, McGraw-Hill, 2005
- **Java™: The Complete Reference, Tenth Edition**, Herbert Schildt, McGraw-Hill, 2005
- **Thinking in Java Fourth Edition** Bruce Eckel, Prentice Hall, 2006
- **Head First Java™, Second Edition**, Kathy Sierra and Bert Bates, O'Reilly, 2005
- **Using Java with 101 Examples**, Atiwong Suchato, First Printing, 2011