Clase incluse - continuare

Programare Orientată pe Obiecte

Nested classes vs *inner classes*

Inner classes:

- datorită relației pe care o au cu clasa exterioară (depind de o instanță a acesteia).
- Cuprind:
  - Clasele interne normale
  - Clasele interne anonime
  - Clasele interne (locale) blocurilor şi metodelor

Nested classes :

- definirea unei clase în interiorul altei clase
- cuprinde atât *inner classes* cât şi *clasele statice interne*.
- clasele statice interne: *static nested classes* (nu *static inner classes)*.

Clase interne statice

- Clasele interne pot avea modificatorul ***static*** (clasele exterioare nu pot fi statice!)
- Putem obține o referință către o clasă internă statică fără a avea nevoie de o instanță a clasei exterioare!
- Diferența clase interne statice şi cele nestatice: **clasele nestatice țin legătura cu obiectul exterior** în vreme ce **clasele statice nu păstrează această legătură**.

Pentru clasele interne statice:

- nu avem nevoie de un obiect al clasei externe pentru a crea un obiect al clasei interne
- nu putem accesa câmpuri nestatice ale clasei externe din clasă internă (nu avem o instanță a clasei externe)

Exemplu

```
class Outer {
    public int data= 9;

    class NonStaticInner {
        private int i = 1;
        public int value() {
         return i + data;   //Outer.this.data;
                        // OK, acces membru  clasă exterioară
        }
    }
     static class StaticInner {
        public int k = 99;
         public int value() {
            k += data; // EROARE,  acces membru nestatic
         return k;
     }
    }
}
```

Exemplu

```
public class Test {
  public static void main(String[] args) {
     Outer out                    = new Outer ();
    Outer.NonStaticInner nonSt = out.new NonStaticInner();
    // instantiere CORECTA pt o clasa nestatica

    Outer.StaticInner st       = out.new StaticInner();
    // instantiere INCORECTA a clasei statice

    Outer.StaticInner st2      = new Outer.StaticInner();
     // instantiere CORECTA a clasei statice
    }
}
```

De ce clase interne statice?

- Pentru a grupa clasele:
  - dacă o clasă internă statică A.B este folosită doar de A, atunci nu are rost să o facem top-level.

- Dacă avem o clasă internă A.B, o declarăm statică dacă în interiorul clasei B nu avem nevoie de nimic specific instanței clasei externe A
  - nu avem nevoie de o instanță a acesteia

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 şi 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

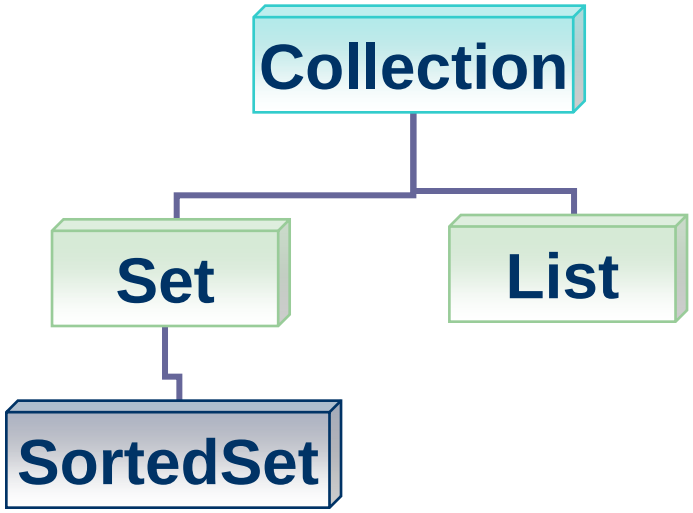
- **Interfețe**: *tipuri abstracte de date* ce descriu colecțiile şi permit utilizarea lor independent de detaliile implementărilor.
- **Implementări**: implementări concrete ale interfețelor ce descriu colecții (clase); reprezintă *tipuri de date reutilizabile*.
- **Algoritmi polimorifici**: metode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

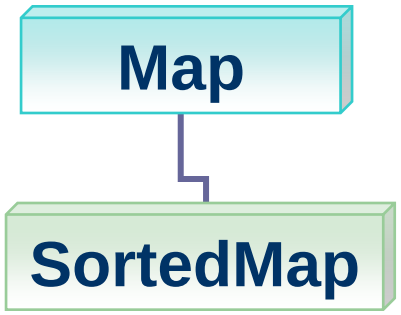
Avantaje:

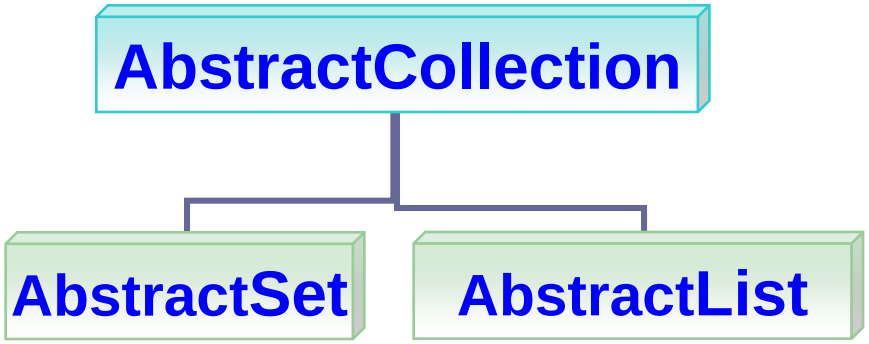
- Reducerea efortului de programare
- Creşterea vitezei şi calității programului

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.





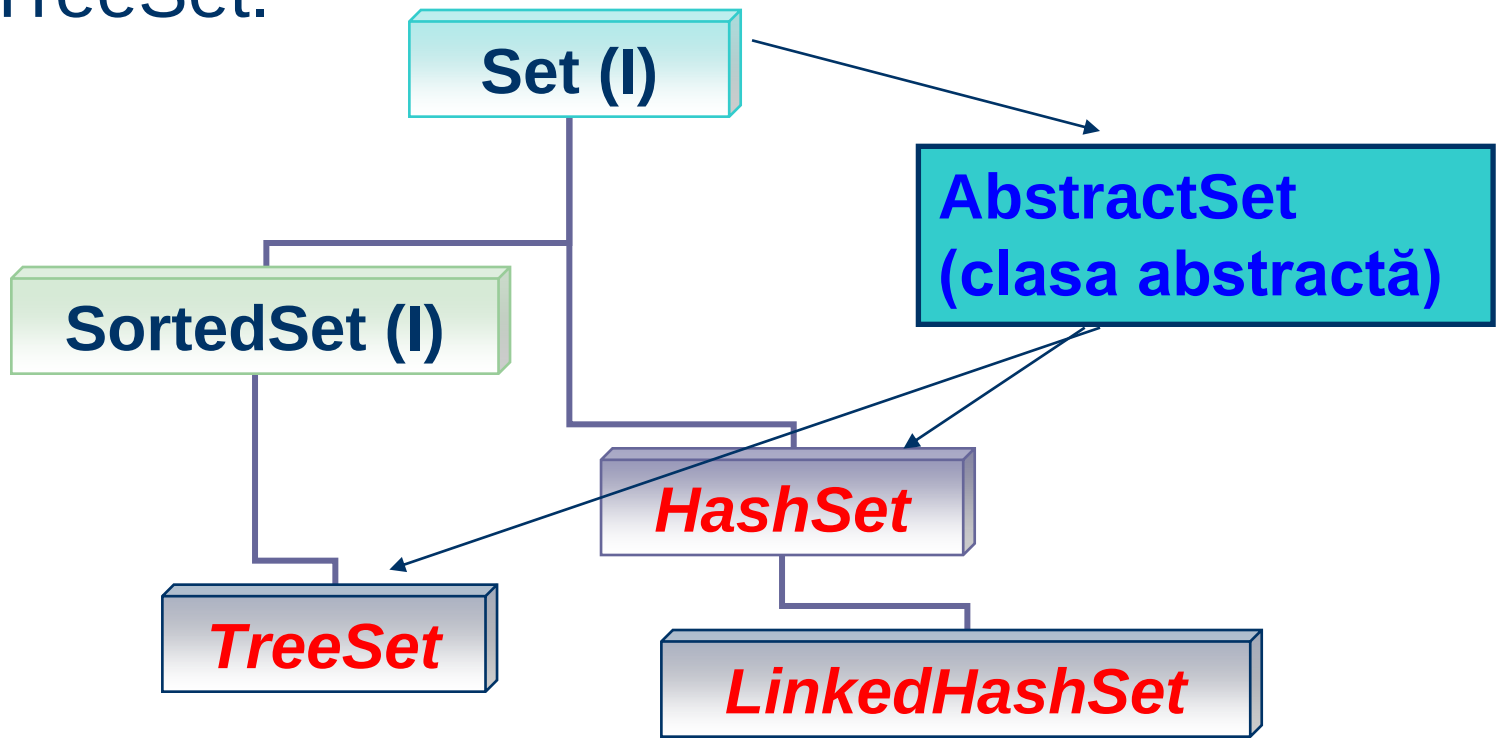


Collection

```
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 şi o2 cu proprietatea o1.equals(o2).
- Implementări: HashSet, LinkedHashSet şi TreeSet.



SortedSet

- mulțime cu elemente sortate.
- ordonarea elementelor se face conform ordinii lor naturale, sau conform cu ordinea dată de un comparator specificat la crearea colecției
- ordinea este menținută automat la orice operație efectuată asupra mulțimii.
- 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 şi 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**

HashSet, LinkedHashSet

HashSet: **tabelă de dispersie** (hash table);

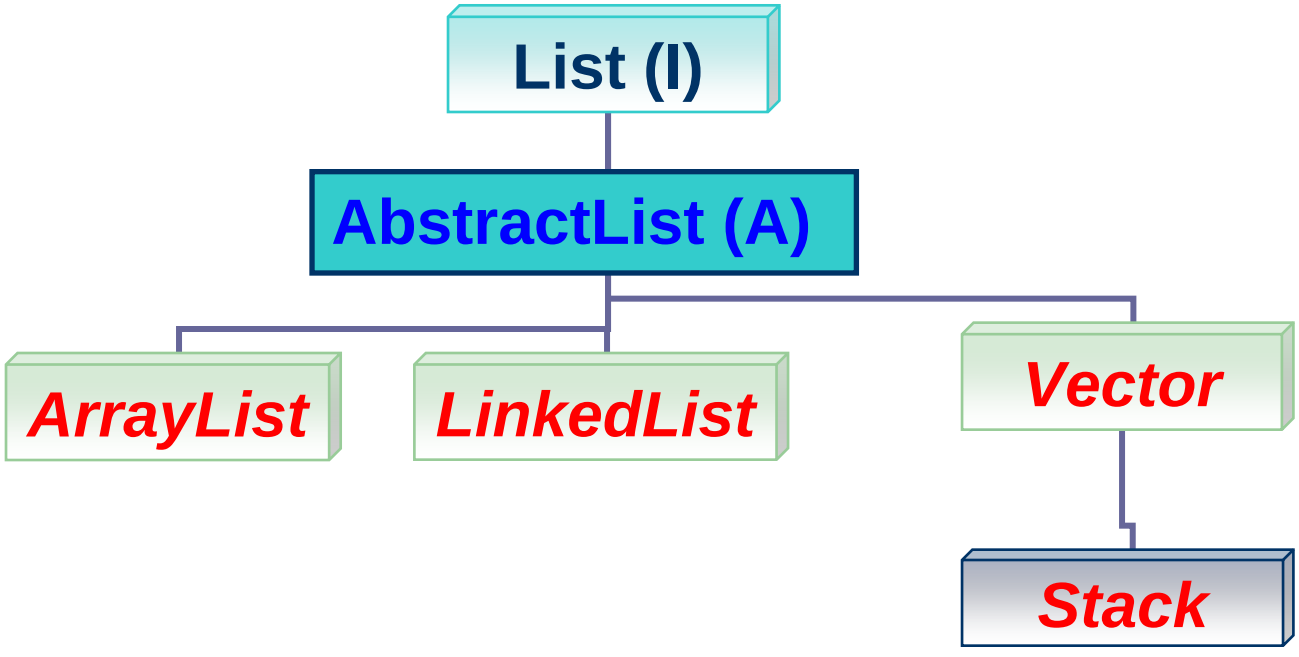
- implementarea cea mai performantă
- calculează codul de dispersie al elementelor pe baza metodei hashCode, definită în clasa Object
- nu avem garanții asupra **ordinii** de parcurgere.
- doi iteratori **diferiți** pot parcurge elementele mulțimii în ordine **diferită**.

LinkedHashSet : tabelă de dispersie.

Diferența față de HashSet - menține o listă dublu-înlănțuită peste toate elementele sale. => elementele rămân în **ordinea** în care au fost inserate.

List

- liste de elemente indexate.
- pot contine duplicate şi permit un control precis asupra poziției unui element prin intermediul indexului acelui element.
- există metode pentru acces pozițional, căutare şi 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

- structuri de tip: cheie – element, ce asociază fiecarui element o cheie unică, după care poate fi regăsit.
- nu pot conține chei duplicate şi 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, LinkedHashMap -** şi **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**

Interfața Map.Entry

- desemnează o pereche (cheie, valoare) din map.
- Metode:
  - **getKey**: întoarce cheia
  - **getValue**: întoarce valoarea
  - **setValue**: permite stabilirea valorii asociată cu această cheie

Implementări ale colecțiilor

< Implementare >< Interfata >

Interfața	Clasa
Set	**HashSet, *LinkedHashSet***
SortedSet	**TreeSet**
List	**ArrayList, LinkedList,Vector, *Stack***
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 aceeaşi funcționalite, cum ar fi ArrayList şi Vector, HashMap şi 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 şi 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 şi 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

- 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.algoritm(colectie,[argumente]);

- majoritatea operează pe liste dar şi pe colecții arbitrare.

Exemple de algoritmi

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

Tipuri generice

- **Tipizarea elementelor unei colecții:**
  TipColectie < 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 şi 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 regăseşte 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 % 2 == 0)  it. set( 0 );
            }
            System . out. print (" Rezultat : " + a);
    }
}
```

Varianta simplificată (1.5)

Atenție!

- 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
    ...
}
```

Exemple

```
Collection<String> c = new ArrayList<String>();
c.add("Test");
c.add(2); // EROARE!
Iterator<String> it = c.iterator();
while (it.hasNext()) {
   String s = it.next();
}
```

- O **iterare** obişnuită pe un map se va face în felul următor:

```
for (Map.Entry<String,Student> entry:
   students.entrySet())
   System.out.println("Media “+ entry.getKey()+“este“
   +entry.getValue().getAverage());
```

- bucla for-each⇔ iteratorul mulțimii de perechi întoarse de entrySet.

Exemplu

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

Rezolvare

```
import java.util.*;

class MyArray {

  private Object v[];
  private int n;

  public MyArray (int nmax) {
        v= new Object[nmax];
        n=0;
  }

  public void add (Object e) {
    if ( n < v.length)
      v[n++]=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);
   }
}
```

Dynamic binding vs static binding

Programare Orientată pe Obiecte

Exercițiu propus

- Cum ar trebui să fie definite clasele Adult, Student şi Inginer astfel încât următoarea secvență să dea eroare la compilare doar unde este specificat?

```
class Test {
   public static void main(String args[]) {
      Adult  a = new Student(); /* fara
                                 eroare */
      Adult b = new Inginer();/* fara
                           eroare */
      a.explorare(); // fara eroare
      b.explorare(); // fara eroare
      a.afisare();   //fara eroare
      b.afisare();   //eroare la compilare!
   }
}
```

Exercițiu propus

```
class Ana {
        public void print(Ana p) {
            System.out.println("Ana 1\n");
        }
}
class Mihai extends Ana {
        public void print(Ana p) {
            System.out.println("Mihai 1\n");
        }
        public void print(Mihai l) {
            System.out.println("Mihai 2\n");
        }
}
class Dana extends Mihai {
        public void print(Ana p) {
            System.out.println("Dana 1\n");
        }
        public void print(Mihai l) {
            System.out.println("Dana 2\n");
        }
        public void print(Dana b) {
            System.out.println("Dana 3\n");
        }
}
```

Exercițiu propus

```
public class Test{
    public static void main (String [] args) {
        Mihai stud1 = new Dana();
        Ana stud2 = new Mihai();
        Ana stud3 = new Dana();
        Dana stud4 = new Dana();
        Mihai stud5 = new Mihai();
 1      stud1.print(new Ana());
 2      ((Dana)stud1).print(new Mihai());
 3      ((Mihai)stud2).print(new Ana());
 4      stud2.print(new Dana());
 5      stud2.print(new Mihai());
 6      stud3.print(new Dana());
 7      stud3.print(new Ana());
 8      stud3.print(new Mihai());
 9      ((Dana)stud3).print(new Mihai());
10      ((Dana)stud3).print(new Dana());
11      stud4.print(new Dana());
12      stud4.print(new Ana());
13      stud4.print(new Mihai());
14      stud5.print(new Dana());
15      stud5.print(new Mihai());
16      stud5.print(new Ana());   } }
```

Ierarhie

Ana – print (Ana)

|

Mihai – print (Ana), print (Mihai)

|

Dana – print (Ana), print (Mihai), print (Dana)

Tip – nume -> obiect

- **Mihai - stud1 -> Dana**
- **Ana - stud2 -> Mihai**
- **Ana - stud3 -> Dana**
- **Dana - stud4 -> Dana**
- **Mihai - stud5 -> Mihai**

Output

1 Dana 1
2 Dana *2*
3 Mihai 1
4 **Mihai 1**
5 **Mihai 1**
6 **Dana 1**
7 **Dana 1**
8 **Dana 1**
9 *Dana 2*
10 *Dana 3*
11 *Dana 3*
12 *Dana 1*
13 *Dana 2*
14 **Mihai 2**
15 Mihai 2
16 Mihai 1

Explicații

1. stud1.print(new Ana())

   stud1 -> *Dana* apelează *Dana.print(Ana*)
2. ((Dana)stud1).print(new Mihai());

   stud1 -> *Dana* apelează *Dana.print(new* Mihai())
3. ((Mihai)stud2).print(new Ana());

   stud2 -> *Mihai* apelează *Mihai.print(new* Ana());
4. stud2.print(new Dana());

   stud2 este declarat *Ana*. Atunci când compilatorul se uită să vadă ce poate apela găseşte metoda *print(Ana*) din clasa *Ana*. La execuție, stud2 este un *Mihai* aşa că va apela metoda p*rint(Ana*) din clasa *Mihai*.
5. Samd