

# Metode avansate de programare

---

## Curs 2

- ☐ Clase interne.
- ☐ Genericitate.
- ☐ Colecții generice de date.

# Clase interne

- Clasele declarate în interiorul unei alte clase se numesc *clase interne (nested classes)*.
- Permit **gruparea claselor** care sunt legate logic și **controlul vizibilității** uneia din cadrul celorlalte.
- Clasele interne sunt de mai multe tipuri, în funcție de modul de a le instanția și de relația lor cu clasa exterioră:
  - **clase interne normale (*regular inner classes*)**
  - **clase anonime (*anonymous inner classes*)**
  - **clase interne metodelor (*method-local inner classes*) sau blocurilor**
  - **clase interne statice (*static nested classes*)**

# Clase interne

## ■ Avantaje:

- Comportamentul acestora ca un membru al clasei;
- O clasă internă poate avea acces la toți membri clasei de care aparține (outer class), inclusiv cei private.
- O clasă internă poate avea modificatorii permisi metodelor și variabilelelor claselor. Astfel, o clasă internă poate fi nu numai public, final, abstract dar și private, protected și static.

# Clase interne normale – declare public

```
public class Outer {
    private int i=1;

    public int value () {
        return i;
    }
    public class Inner {
        private int i;
        public Inner (int i) {
            this.i = Outer.this.i+i;
            Outer.this.i=this.i+1;
        }
        public int value () {
            return i;
        }
    }

    public Inner getInnerInstance () {
        return new Inner(1);
    }
}
```

```
public class TestInner {

    public static void main(String[] args) {

        Outer out = new Outer();

        Outer.Inner in1 = out.getInnerInstance();
        Outer.Inner in2 = out.new Inner(10);

        System.out.println(in1.value());
        System.out.println(in2.value());
        System.out.println(out.value());
    }
}
```

- Pentru a instanția Inner, avem nevoie de o instanța Outer!
- Referința la clasa externă în Inner: **Outer.this**

# Clase interne normale – declare protected

```
public class Outer {  
    private int i=1;  
  
    public int value () {  
        return i;  
    }  
    protected class Inner {  
        private int i;  
        public Inner (int i) {  
            this.i = Outer.this.i+i;  
            Outer.this.i=this.i+1;  
        }  
        public int value () {  
            return i;  
        }  
    }  
  
    public Inner getInnerInstance () {  
        return new Inner(1);  
    }  
}
```

```
public class TestInner {  
  
    public static void main(String[] args) {  
  
        Outer out = new Outer();  
  
        Outer.Inner in1 = out.getInnerInstance();  
        Outer.Inner in2 = out.new Inner(10);  
  
        System.out.println(in1.value());  
        System.out.println(in2.value());  
        System.out.println(out.value());  
    }  
}
```

- La fel cu cele declate public, doar ca vizibilitatea este la nivel de pachet!!! TestInner este in acelasi pachet cu clasa Outer!

# Clase interne normale – declarate private

```
public interface Hidden {
    public int value ();
}

public class Outer {
    private class HiddenInner implements Hidden {
        private int i;
        public HiddenInner (int i) {
            this.i = i;
        }
        public int value () {
            return i;
        }
    }

    public Hidden getInstance () {
        return new HiddenInner(10);
    }
}
```

```
public static void main(String[] args) {

    Outer out = new Outer();
    Hidden in3 = out.getInnerInstance();
    System.out.println(in3.value());
}
```

- Definierea interfeței Hidden este utilă pentru a putea asocia clasei HiddenInner un tip, care să ne permita folosirea instanțelor acesteia, altfel tipul ei nu ar fi fost vizibil pentru că a fost declarată private.

```
Outer.HiddenInner in1 = out.getInnerInstance(); // eroare
Outer.HiddenInner in2 = new Outer().new HiddenInner(i:10); // eroare
```

'ClaseInterne.normale.Outer.HiddenInner' has private access in 'ClaseInterne.normale.Outer'

# Clase interne în metode

```
public interface Hidden {
    public int value ();
}

public class Outer {
    public Hidden getInnerInstance() {
        int a=9;
        class FuncInner implements Hidden {
            private int i = 11;
            public int value() {
                i+=a;
                //a++;
                return i;
            }
        }
        return new FuncInner();
    }
}
```

```
public class TestInnerMetode {
    public static void main(String []args)
    {
        Outer out = new Outer();
        Hidden in3 = out.getInnerInstance();
        System.out.println(in3.value());
    }
}
```

- Clasele interne declarate în metode nu pot folosi variabilele declarate în metoda respectivă și nici parametrii metodei.
- Pentru a putea accesa variabilele declarate în metoda respectivă, acestea trebuie declarate **final sau effectively final**

# Clase interne în blocuri

```
public interface Hidden {  
    public int value ();  
}
```

```
class Outer {  
    public Hidden getInnerInstance(int i) {  
        if (i == 11) {  
            class BlockInner implements Hidden {  
                private int i = 11;  
                public int value() {  
                    return i;  
                }  
            }  
            return new BlockInner();  
        }  
        return null;  
    }  
}
```

- La compilare clasa va fi creată indiferent care este valoarea de adevăr a condiției *if*.



# Clase interne anonime

```
public interface Hidden {  
    public int value ();  
}
```

```
class Outer {  
    public Hidden getInnerInstance(int i) {  
        return new Hidden() {  
            private int i = 11;  
            public int value() {  
                return i;  
            }  
        };  
    }  
}
```

```
Outer out = new Outer();
```

```
Hidden in3 = out.getInnerInstance(11);
```

```
System.out.println(in3.value());
```

# Clase interne statice

```
package ClaseInterne.statice;
public class Outer {
    public int outerMember = 9;
    public class NonStaticInner {
        private int i = 1;
        public int value() {
            return i + Outer.this.outerMember; // OK, putem accesa un membru al clasei exterioare
        }
    }
    public static class StaticInner {
        public int k = 99;
        public int value() {
            //k += outerMember; // EROARE, nu putem accesa un membru nestatic al clasei exterioare
            return k;
        }
    }
}
```

`Outer.StaticInner staticInnerClass = new Outer.StaticInner();`

Pentru a instanția o clasă statică nu avem nevoie de o instanță Outer!

# Clase interne concluzii

Clase interne normale -publice -private -protected (accesibile la nivel de pachet)	<pre>Outer out = new Outer (); Outer.Inner in2 = out.new Inner(10);  Hidden in1 = out.getInnerInstance();</pre>
Clase interne anonime	<pre>Outer out = new Outer(); Hidden in3 = out.getInnerInstance();</pre>
Clase interne în metode și blocuri	<pre>Outer out = new Outer(); Hidden in3 = out.getInnerInstance();</pre>
Clase interne statice	<pre>Outer.StaticInner st2 = new Outer.StaticInner();</pre>

# Genericitate

## ■ Problemă:

- Construiți o structură de date: *o stivă, o listă înlănțuită, un vector, un graf, un arbore, etc.*
- Care este tipul de date pe care îl vom folosi pentru reprezentarea elementelor?

```
class Stack {  
    private int[] items;  
    int vf;  
    public Stack(){...}  
    public void push (int item) {...}  
    public int peek() {...}  
}
```

```
Stack stack = new Stack();  
stack.push(100);  
stack.push(200);  
stack.push("Hello World!");
```

```
public class Stack {  
    private Object[] items;  
    private int vf=0;  
    public Stack(){...}  
    public void push (Object item) {...}  
    public Object peek() {...}  
}
```

```
Stack stack = new Stack();  
stack.push(100);  
stack.push(new Rectangle());  
stack.push("Hello World!");  
String s = (String) stack.peek();
```

Cast ????

# Genericitate

```
public class Stack {  
    private Object[] items;  
    private int vf=0;  
    public Stack(){...}  
    public void push (Object item) {...}  
    public Object peek() {...}  
}
```

```
//testStack.java
```

```
Stack stack = new Stack();  
stack.push(100);  
stack.push(new Rectangle());  
stack.push("Hello World!");
```

```
while (!stack.isEmpty())  
{  
    String s = (String)stack.pop();  
    System.out.println(s);  
}
```

//corect la compilare, eroare la executie

# Tipuri generice

- Permit parametrizarea tipurilor de date (clase și interfețe), parametri fiind tot tipuri de date
- Au fost introduse începând cu **versiunea 1.5**
- Asemănătoare cu template din C++
  - *Diferențe majore:*
    - nu se creează o nouă clasă pentru fiecare tip;
    - pot fi impuse constrângeri pentru tipul generic.
- *Motivația:*
  - Vezi exemplu anterior

# Declararea claselor

```
[modifier_acces] class NumeClasa <TipGeneric1 [, TipGeneric2[,...]] {  
    [modifier_acces] TipGeneric1 atribut1;  
    declaratii de attribute si metode  
    declaratii si definitii de metode  
}
```

## ■ Convenții de numire a tipurilor

- E - Element (folosit intensiv de Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

Exemple:

```
public class Stack<E> { ... }  
public class Node<T> { ... }  
public interface Pair<K, V> { ... }  
public class PairImpl<K, V> implements Pair<K, V> {...
```

# Instanțierea tipurilor generice

```
public class Stack<E>{  
    public void push(E elem) {...}  
    public E peek() {...}  
}
```

```
public static void main(String[] args)  
{  
    //test Stack<E>  
    Stack<String> ss=new Stack<String>();  
    ss.push("Ana");  
    ss.push("Maria");  
    //ss.push(new Persoana("Ana", 23)); //eroare la compilare  
    String elem=ss.peek(); //fara cast  
    Stack<Persoana> sp=new Stack<Persoana>();  
    sp.push(new Persoana("Ana", (byte)23));  
    sp.push(new Persoana("Maria", (byte)10));  
}
```



# Instanțierea tipurilor generice

- Când se creează instanțe ale unor clase generice, nu se pot folosi tipurile primitive: int, byte, char, float, double,....
- Se folosesc clasele asociate:

Tip primitiv	Clasa învelitoare
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

```
Stack<int> si=new Stack<int>(); //eroare la compilare  
Stack<Integer> si=new Stack<Integer>();
```

# Autoboxing

```
public class Stack<E>{  
    public void push(E elem) {...}  
    public E peek() {...}  
}
```

```
Stack<Integer> si=new Stack<Integer>();  
si.push(23); //conversie automata  
si.push(new Integer(23));  
int val=si.peek();
```

- Se convertește automat o dată de tip primitiv într-un obiect al clasei asociate (când compilatorul așteaptă un obiect) și invers (când se așteaptă un tip primitiv).

# Tablouri cu elemente de tip generic

```
3 public class Stack<E> {  
4     private E[] items=new E[20];  
5  
6
```

Type parameter 'E' cannot be instantiated directly

- Nu pot fi create folosind operatorul **new**

- Doar:

```
T[] items=(T[])new Object[20]; //warning la compilare
```

# Tablouri cu elemente de tip generic - Alternative

- Se creeaza un tablou cu elemente de tip **Object**, iar cand se cere un element se face cast explicit:

```
public class Stack<E> {  
    Object[] items=new Object[20];  
    int vf=0;  
  
    . . .  
    public E peek() {  
        if (vf>0)  
            return (E)items[vf-1];  
        else return null;  
    }  
}
```


# Tablouri cu elemente de tip generic - Alternative

- Se foloseste metoda **Array.newInstance** - urmeaza cursul de Reflectie in Java

```
public class Stack<E> {  
  
    E[] items;  
    int vf=0;  
    public Stack(Class<E> tip)  
    {  
        items=(E[]) Array.newInstance(tip, 20);  
    }  
    public void push (E item) {  
        if (vf<items.length){  
            items[vf++]=item;  
        }  
    }  
    . . .  
}
```

**import** java.lang.reflect.Array;

Stack<Integer> st=new Stack<>(Integer.class);  
st.push(2);  
st.push(3);  
System.out.println(st.peek());



- Se foloseste clasa **ArrayList** in loc de tablou

# Metode generice

- Exemplu 1: Metodele generice pot fi definite in clase care nu declara tipuri generice.

```
class Util {
    public static <T> int countNullValues(T[] anArray){
        int count = 0;
        for (T e : anArray)
            if (e == null) {
                ++count;
            }
        return count;
    }
}

public static void main(String[] args)
{
    //apel metoda generica
    int k= Util.countNullValues(new String[]{"a", null, "b"});
    int j= Util.countNullValues(new Integer[]{1, 2, null, 3, null});
}
```

# Metode generice

- Exemplu 2: Tipul generic al unei metode generice poate sa fie diferit de tipul generic al clasei in care e declarata metoda generica.

```
public class Stack<E> {  
    E[] items;  
    int vf=0;  
  
    public static <T> void copiaza(T[] elems, Stack<T> st) {  
        for(T e:elems)  
            st.push(e);  
    }  
}
```

```
Stack<Integer> st=new Stack<>(Integer.class);  
st.push(2);  
st.push(3);
```

```
Stack.copiaza(new Integer[]{4,5,6},st);  
System.out.println(st.peek());
```

# Membri statici în contextul tipurilor generice

```
public class Singleton<T> {  
  
    public static T getInstance() {  
        if (instance == null)  
            instance = new Singleton<T>();  
  
        return instance;  
    }  
  
    private static T instance = null;  
}
```

- NU putem avea attribute statice de tip generic!



# Erase

- In Java, nu se creeaza o noua clasa pentru fiecare instanta a unei clase generice (cu tip diferit).
- La compilare, compilatorul “sterge” (erases) informatiile despre tipul generic, inlocuieste fiecare variabila de tip generic cu limita superioara a tipului (de obicei Object), si unde este nevoie insereaza un cast explicit catre tipul generic.
- Motivul: compatibilitatea cu versiunile anterioare de Java, cand nu exista generics.
- Nu se recompileaza clasa generica pentru fiecare instanta noua (C++).

# Tipuri generice delimitate (bounded)

- Se pot specifica constrangeri (limite) pentru tipul generic, folosind cuvântul **extends**.  
    **T extends [C &] I1 [& I2 &...& In]**  
    T mosteneste clasa C si implementeaza interfetele I1, ... In.
- Cand se specifica constrangeri, la compilare T este inlocuit cu primul element din expresia de constrangeri.  
    **T extends C //T este inlocuit cu C**  
    **T extends C & I1 & I2 //T este inlocuit cu C**  
    **T extends I1 & I2 //T este inlocuit cu I1**  
    **T extends I1 //T este inlocuit cu I1**  
    **T //T este inlocuit cu Object**
- Daca se specifica constrangeri pentru tipul T, atunci folosind o variabila de tipul T se poate apela orice metoda din clasa sau interfetele precizate ca limita.

# Tipuri generice delimitate (bounded)

```
interface Comparable<E>{
    int compareTo(E element);
}

public class ListaOrdonata<E extends Comparable<E>>
implements Lista<E> {
    private class Nod{
        E info;
        Nod urm;
        Nod(E elem){info=elem; urm=null;}
        Nod(E info, Nod urm){
            this.info=info;
            this.urm=urm;
        }
    }
    private Nod cap;
    private int count=0;
    public ListaOrdonata(){ cap=null;}
```

```
public void adauga(E elem){
    count++;
    if (cap==null){
        cap=new Nod(elem);
        return;
    }
    if (elem.compareTo(cap.info)<0){
        cap=new Nod(elem, cap);
    }else {
        Nod p=cap;
        Nod r=p;
        while (p!=null && p.info.compareTo(elem)<0)
        {
            r=p;
            p=p.urm;
        }
        . . .
    }
}
```

# Tipuri generice delimitate (bounded)

```
public class Persoana implements Comparable<Persoana>{
    protected String nume;
    protected int varsta;
    public Persoana(){
        this("",0);
    }
    public Persoana(String nume, int varsta){
        this.nume=nume;
        this.varsta=varsta;
    }

    @Override
    public int compareTo(Persoana o) {
        return this.nume.compareTo(o.nume);
    }
    . . .
}

public static void main(String[] args)
{
    Persoana p=new Persoana("Pop",12);
    Persoana p2=new Persoana("Ion",9);
    Persoana p3=new Persoana("Dan",14);
    ListaOrdonata<Persoana> l=new ListaOrdonata<Persoana>();
    l.adauga(p);
    l.adauga(p2);
    l.adauga(p3);
    for (int i=0; i<l.size(); i++)
        System.out.println(l.get(i));
}
```

# Problemă

```
public class Student extends Persoana {  
    private static final int COD_UNIVERSITATE = 15;  
    private int anStudiu;  
}
```

```
ListaOrdonata<Student> studList=new ListaOrdonata<Student>();
```

Type parameter 'Generics.Student' is not within its bound; should implement 'java.lang.Comparable<Generics.Student>'

## ■ Soluția:

```
public class ListaOrdonata<E extends Comparable<? super E>> implements Lista<E> {
```

Adică:

Constrângerea `<E extends Comparable<E>>` va fi relaxată cu `<E extends Comparable<? super E>>`

Ceea ce înseamnă că E este un tip comparabil cu el, dar și cu orice alt tip de bază al său.

# Genericitatea în subtipuri

```
ListaOrdonata<Persoana> personList;  
ListaOrdonata<Student> studList=new ListaOrdonata<Student>();  
personList=studList; // ??????????  
personList.adauga(new Persoana("Anda",12));
```



Dacă *ChildType* este un subtip (clasă descendentă sau subinterfață) al lui *ParentType*, atunci o structură generică *GenericStructure<ChildType>* **nu** este un subtip al lui *GenericStructure<ParentType>*.

- Soluția: Wildcards
- `ListaOrdonata<?> lista;`  
  
`lista=studList;`

# Wildcards

- Wildcard-urile sunt utilizate atunci când dorim să întrebuițăm o structură de date generică (*parametru* într-o funcție) și nu dorim să limităm tipul de date din colecția respectivă

```
public static void main(String[] args)
{
    Persoana p=new Persoana("Pop",12);
    Persoana p2=new Persoana("Ion",9);
    Persoana p3=new Persoana("Dan",14);

    ListaOrdonata<Persoana> personList=new ListaOrdonata<Persoana>();
    ListaOrdonata<Student> studList=new ListaOrdonata<Student>();

    personList.adauga(p); personList.adauga(p2);
    personList.adauga(p3);

    studList.adauga(p); studList.adauga(p2); studList.adauga(p3);

    printCollection2(personList);
    printCollection2(studList);
}

public static void printCollection2(ListaOrdonata<?> c)
{
    for(int i=0; i<c.size();i++)
        System.out.println(c.get(i));
}
```

# Wildcards

- Limitare: **nu putem adăuga elemente arbitrare** într-o colecție cu wildcard-uri:

```
ListaOrdonata<?> c = new ListaOrdonata<String>(); // Operatie permisa  
c.adauga(23); // Eroare la compilare
```

**Soluția:** ((ListaOrdonata<String>)c).adauga("Some String");

- Eroarea apare deoarece nu putem adăuga într-o colecție generica decât elemente **de un anumit tip**, iar wildcard-ul **nu indică un tip anume**.



# Wildcard-uri delimitate superior

- Daca se specifica o limita superioara pentru ?, se pot apela metode apartinand clasei sau interfetei din limita superioara.

```
public static void printCollection(ListaOrdonata<? extends Persoana> c)
{
    for(int i=0; i<c.size();i++)
        System.out.println(c.get(i));
}
```

```
ListaOrdonata<Student> studList=new
ListaOrdonata<Student>();
```

```
Student s=new Student("x",12,3);
Student s1=new Student("a",14,1);
Student s3=new Student("z",9,2);
studList.adauga(s);
studList.adauga(s1);
studList.adauga(s3);
```

```
printCollection(studList);
```

# Wildcard-uri delimitate inferior

```
public static void printCollection1(ListaOrdonata<? super Student> c)
{
    for(int i=0; i<c.size();i++)
        System.out.println(c.get(i));
}
```

```
Persoana p=new Persoana("Pop",12);
Persoana p2=new Persoana("Ion",9);
Persoana p3=new Persoana("Dan",14);
```

```
ListaOrdonata<Persoana> personList=new
ListaOrdonata<Persoana>();
printCollection1(personList);
```

- <http://docs.oracle.com/javase/tutorial/java/generics/QandE/generics-questions.html>

# Cadrul colecțiilor Java (Java Collections Framework (JFC))

- O *colecție* este un obiect care grupează mai multe elemente într-o singură unitate. (ex. *Vetori, liste înlănțuite, stive, mulțimi matematice, dicționare, tabele hash, etc.*)
- Reutilizarea codului
- Reducerea efortului de programare
- Creșterea vitezei și calității unei aplicații
- Algoritmi *polimorfici*
- Folosesc *tipuri generice*

# Arhitectura colecțiilor

Interfață



Clasă abstractă



Implementări concrete

List

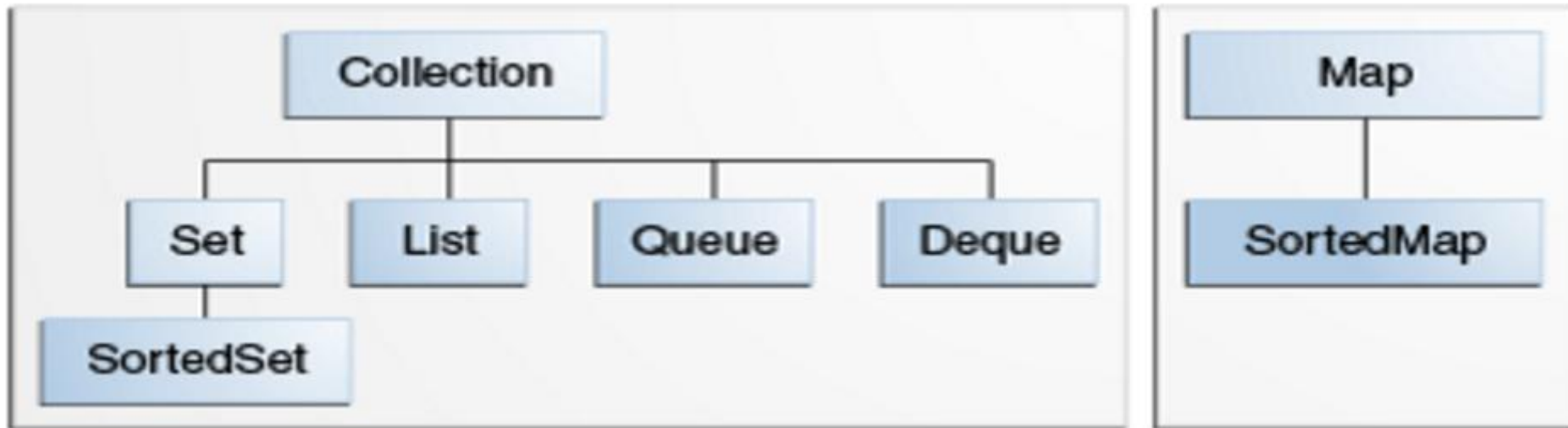


AbstractList



ArrayList  
LinkedList  
Vector

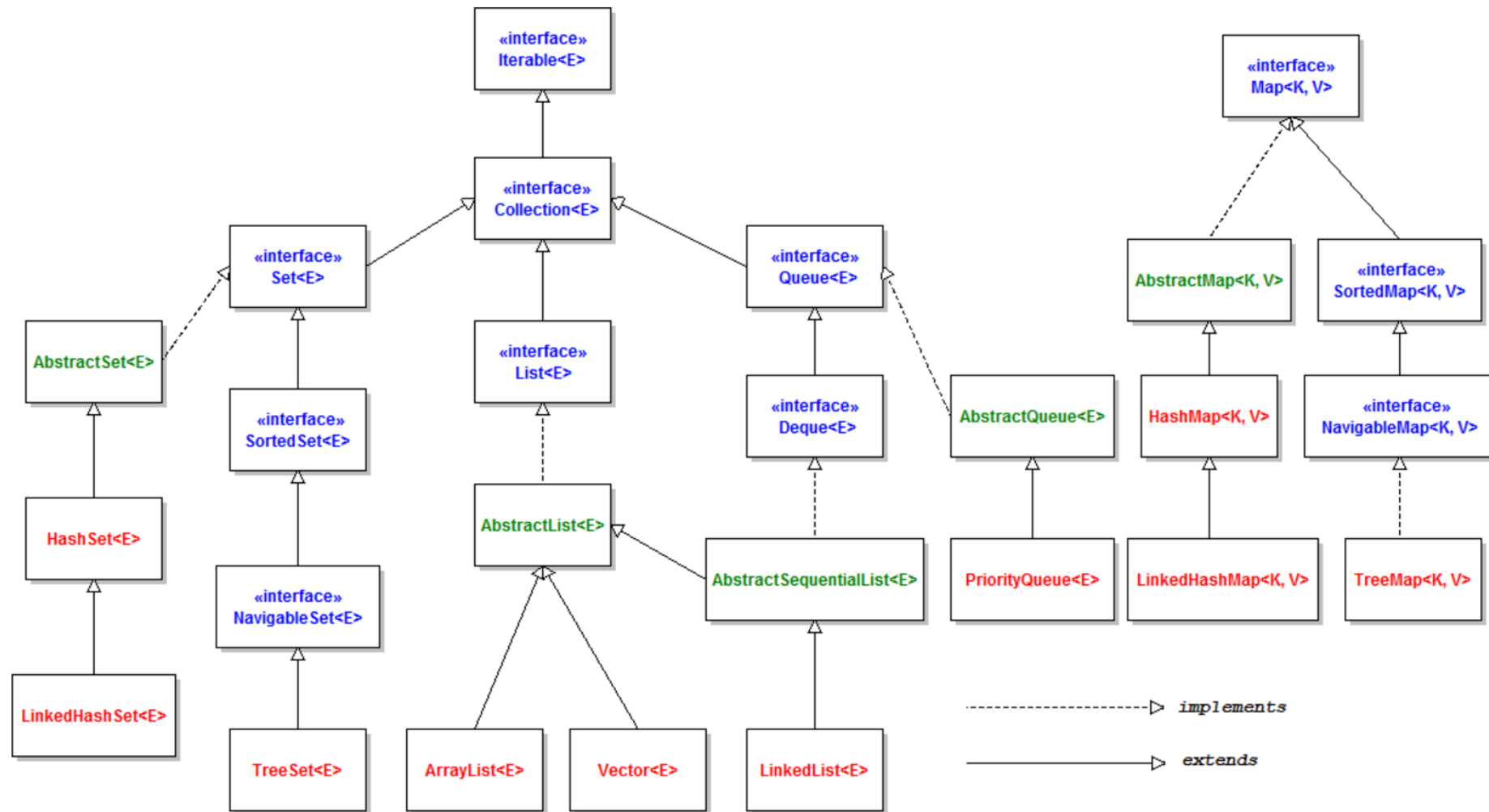
# Interfețe



# Implementari -reprezentare

Interfața	Hash	Array	Tree	Linked	Hash+Linked
<b>Set</b>	HashSet		TreeSet		LinkedHashSet
<b>List</b>		ArrayList Vector		LinkedList	
<b>Queue</b>					
<b>Deque</b>		ArrayDeque		LinkedDeque	
<b>Map</b>	HashMap Hashtable		TreeMap		LinkedHashMap

# Ierarhia colecțiilor



Class diagram of Java Collections framework

# Declarare și instanțiere obiecte

```
Set set = new HashSet(); //--raw generic type (Object)
ArrayList<Integer> list = new ArrayList<>();
List<Integer> list = new ArrayList<>();
List<Integer> list = new LinkedList<>();
List<Integer> list = new Vector<>();
Map<Integer, String> map = new HashMap<>();
```



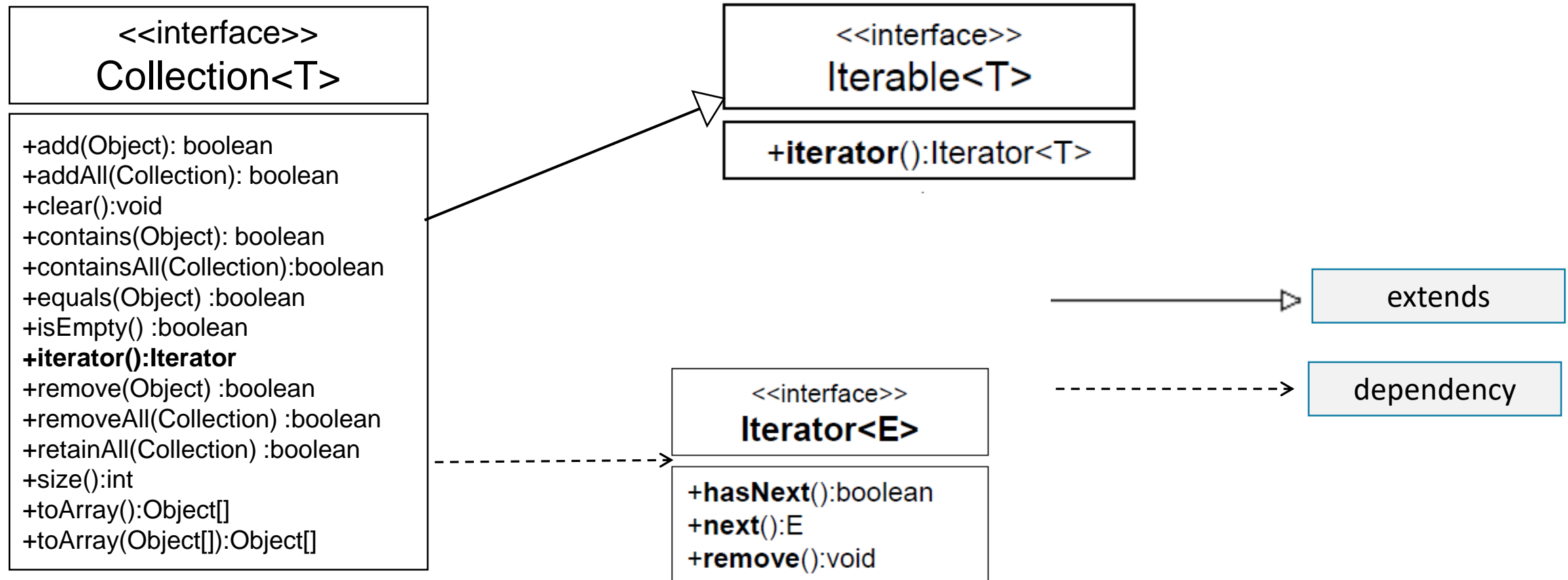
# Interfața Collection

<<interface>>  
Collection<T>

+add(Object): boolean  
+addAll(Collection): boolean  
+clear():void  
+contains(Object): boolean  
+containsAll(Collection):boolean  
+equals(Object) :boolean  
+isEmpty() :boolean  
**+iterator():Iterator**  
+remove(Object) :boolean  
+removeAll(Collection) :boolean  
+retainAll(Collection) :boolean  
+size():int  
+toArray():Object[]  
+toArray(Object[]):Object[]

# java.util.Iterator<E>, java.lang.Iterable<T>

- **Colectiile sunt iterable:** Interfata Collection<E> implementeaza interfata Iterable<E>;



# Parcurgerea colecțiilor

- for-each

```
public void list(Collection<T> items) {
```

```
    for (Item item : items) {  
        System.out.println(item);  
    }
```

=

- Cu iterator

```
public void list(Collection<T> items) {
```

```
    Iterator<Item> it = items.iterator();  
    while(it.hasNext()) {  
        Item item = it.next();  
        System.out.println(item);  
    }
```

# java.util.Collections

- Clasa **Collections** contine exclusiv **metode statice** pentru sortarea unei colectii, pentru determinarea minimului, maximului, inversarea unei colectii, cautarea unei valori, etc.

# Interfața List

- O listă este o colecție **ordonată**. Listele pot conține elemente **duplicate**.
- Pe lângă operațiile moștenite de la **Collection**, interfața **List** definește următoarele operații:
  - `T get(int index)` - întoarce elementul de la poziția `index`
  - `T set(int index, T element)` - modifică elementul de la poziția `index`
  - `void add(int index, T element)` - adaugă un element la poziția `index`
  - `T remove(int index)` - șterge elementul de la poziția `index`
- List posedă două implementări standard:
  - **ArrayList** - implementare sub formă de vector. Accesul la elemente se face în timp constant: **O(1)**
  - **LinkedList** - implementare sub formă de listă dublu înlănțuită. Prin urmare, accesul la un element nu se face în timp constant, fiind necesară o parcurgere a listei: **O(n)**.
- Printre algoritmi implementați se numără:
  - *sort* - realizează sortarea unei liste
  - *binarySearch* - realizează o căutare binară a unei valori într-o listă

# Interfața Set

- Un Set (mulțime) este o colecție ce nu poate conține elemente duplicate.
- Interfața Set **conține doar metodele moștenite din Collection**, la care adaugă restricții astfel încât elementele duplicate să nu poată fi adăugate.
- Avem trei implementări utile pentru Set:
  - **HashSet**: memorează elementele sale într-o **tabelă de dispersie** (hash table); **este implementarea cea mai performantă**, însă nu avem garanții asupra ordinii de parcurgere. **Doi iteratori diferiți pot parcurge elementele mulțimii în ordine diferită.**
  - **TreeSet**: memorează elementele sale sub formă de **arbore roșu-negru**; elementele sunt ordonate pe baza valorilor sale. Implementarea este mai lentă decât HashSet.
  - **LinkedHashSet**: este implementată ca o **tabelă de dispersie**. Diferența față de HashSet este că LinkedHashSet menține o **listă dublu-înlănțuită** peste toate elementele sale. Prin urmare elementele rămân în ordinea în care au fost inserate. O parcurgere a LinkedHashSet va găsi elementele mereu în această ordine.

**Vezi Seminar 3 – metodele hashCode, equals din clasa Object!**

# Interfața Map

- Un Map este un obiect care mapează chei la valori. Într-o astfel de structură nu pot exista chei duplicate.
- *Fiecare cheie este mapată la exact o valoare.*
- *Map reprezintă o modelare a conceptului de funcție: primește o entitate ca parametru (cheia), și întoarce o altă entitate (valoarea).*
- Cele trei implementări pentru Map sunt:
  - HashMap
  - TreeMap
  - LinkedHashMap
- Particularitățile de implementare corespund celor de la Set;

# Compararea elementelor

- Pentru compararea a doua elemente dintr-o colectie avem doua posibilitati:

1) Entitatile colectiei **implementeaza** interfata **java.lang.Comparable<T>**

```
public interface Comparable<T>{  
    int compareTo(T o);  
}
```

2) Definim un **Comparator** care **implementeaza** **java.util.Comparator<T>**

```
public interface Comparator<T>{  
    int compare(T o1, T o2);  
}
```

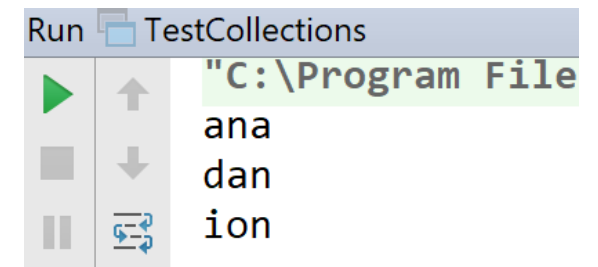


# Compararea elementelor - Comparable<T>

- Obiectele de tip Persoana sunt comparabile – via metoda **compareTo!**

```
public class Persoana implements Comparable<Persoana>{  
    protected String nume;  
    protected int varsta;  
    . . .  
    @Override  
    public int compareTo(Persoana o) {  
        return this.nume.compareTo(o.nume);  
    }  
}
```

```
TreeSet<Persoana> pSet = new TreeSet<>();  
pSet.add(new Persoana("dan", 10));  
pSet.add(new Persoana("ana", 10));  
pSet.add(new Persoana("ion", 10));  
  
for (Persoana p : pSet) {  
    System.out.println(p.getNume());  
}
```



# Compararea elementelor – Comparator<T>

- Clasa Persoana nu implementează interfața Comparable<T>, dar ....

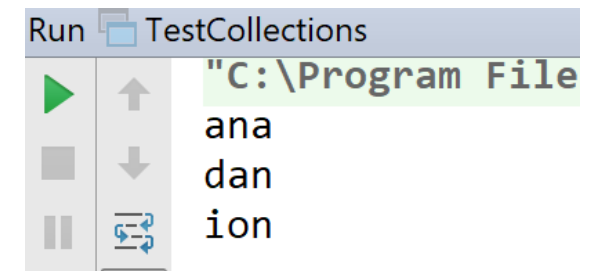
```
public class Persoana { // implements Comparable<Persoana>{
    protected String nume;
    protected int varsta;
    public Persoana(){
        this("",0);
    }
    public Persoana(String nume, int varsta){
        this.nume=nume;
        this.varsta=varsta;
    }

    // @Override
    // public int compareTo(Persoana o) {
    //     return this.nume.compareTo(o.nume);
    // }
}
```

# Compararea elementelor – Comparator<T>

- Instantiem un comparator (aici clasă internă anonimă) ce va fi referit in constructorul clasei TreeSet.

```
public static void main(String[] args) {  
  
    TreeSet<Persoana> pSet = new TreeSet<>(new Comparator<Persoana>() {  
        @Override  
        public int compare(Persoana o1, Persoana o2) {  
            return o1.getNume().compareTo(o2.getNume());  
        }  
    });  
  
    pSet.add(new Persoana("dan", 10));  
    pSet.add(new Persoana("ana", 10));  
    pSet.add(new Persoana("ion", 10));  
  
    for (Persoana p : pSet) {  
        System.out.println(p.getNume());  
    }  
}
```



# Sortarea unei liste - exemplu

```
public class Persoana implements Comparable<Persoana>{  
    protected String nume;  
    protected int varsta;  
    . . .  
}
```

```
List<Persoana> persons=new ArrayList<>();
```

```
persons.add(new Persoana("dan", 30));  
persons.add(new Persoana("ana", 89));  
persons.add(new Persoana("ion", 32));
```

```
//sorteaza lista persoane dupa varsta  
persons.sort(new ComparatorPersoanaVarsta());  
System.out.println("Lista sortata dupa varsta");  
for (Persoana p : persons) { System.out.println(p);}
```

```
// sorteaza lista persoane dupa nume  
Collections.sort(persons); //Persoana implementeaza Comparable<Persoana>  
System.out.println("Lista sortata dupa varsta");  
for (Persoana p : persons) { System.out.println(p);}
```

# Map - example

```
Map<Integer,Persoana> map=new LinkedHashMap<>();
map.put(1, new Persoana("dan", 30));
map.put(2, new Persoana("ana", 89));
map.put(3, new Persoana("ion", 32));

for(Integer key: map.keySet())
    System.out.println(map.get(key));

for(Map.Entry<Integer,Persoana> entry : map.entrySet()
)
    System.out.println(entry.getValue());
```

- Vezi sem 3 pt mai multe exemple

# Cursul următor

- Excepții
- I/O