Legare statică/dinamică – static/dynamic binding în Java

Programare Orientată pe Obiecte

Polimorfism

- **Polimorfism – abilitatea unui obiect de a se comporta diferit la acelaşi mesaj**
- **Două tipuri: static şi dinamic**

- **Supraîncărcarea (overloading)**
- **Supradefinirea (overriding)**

```
class A {
    void metoda() {
        System.out.println("A: metoda fara parametru");
    }
    // Supraîncărcare
    void metoda(int arg) {
        System.out.println("A: metoda cu un parametru");
    }
}
class B extends A {
    // Supradefinire
    void metoda() {
        System.out.println("B: metoda fara parametru");
    }
}
```

Static/dynamic binding în Java

- Binding – procesul de a stabili ce metodă sau variabilă va fi apelată

- **Static binding şi dynamic binding - două concepte importante în Java**
- Legate direct de execuția codului
- Mai multe metode cu acelaşi nume (method overriding) sau două variabile cu acelaşi nume în aceeaşi ierarhie de clase, care este utilizată?
- Majoritatea referințelor sunt rezolvate în timpul compilării, dar cele care depind de obiect şi polimorfism sunt rezolvate la execuție, atunci când este de fapt disponibil obiectul.

- Dynamic binding – late binding (la execuție)
- Static binding – early binding (la compilare)

Static/dynamic binding în Java

- *Diferența între static şi dynamic binding* în Java
- Întrebare populară la angajarea în domeniu
- Explorează cunoştințele candidaților legate de cine determină ce metodă va fi apelată dacă există mai multe metode cu acelaşi nume, ca în cazul metodelor supraîncărcate sau supradefinite (method overloading and overriding).
- Compilatorul sau JVM – maşina virtuală Java?

Legare statică/dinamică – Static/dynamic binding în Java

```
class Vehicle {
    public void drive() {
        System.out.println("A");
    }
}

class Car extends Vehicle {
    public void drive() {
        System.out.println("B");
    }
}
class TestCar {
    public static void main(String args[]) {
        Vehicle v;
        Car c;
        v = new Vehicle();
        c = new Car();
        v.drive();
        c.drive();
        v = c;
        v.drive();
    }
}
```

Output

A

B

B

Polimorfism

- Legarea dinamică are loc în Java pentru orice metodă care nu are atributul *final* sau *static*, metodă numită polimorfică.
- In Java majoritatea metodelor sunt polimorfice.
- Metodele polimorfice Java corespund functiilor virtuale din C++.
- Apelul functiilor polimorfice este mai putin eficient ca apelul functiilor cu o singura formă.
- Fiecare clasă are asociat un tabel de pointeri la metodele (virtuale) ale clasei.

Static Binding vs Dynamic binding în Java

Diferențe între legarea statică şi cea dinamică în Java:

- **Static binding** se realizează la **Compilare** în timp ce **Dynamic binding** se realizează la **Rulare** - Execuție.

- Metodele şi variabilele **private**, **final** sau **static** utilizează static binding în timp ce metodele **virtuale - abstracte** utilizează dynamic binding.

- **Static binding** se face pe baza informației legate de **Tip** (**clasa** în Java), în timp ce **Dynamic binding** utilizează **Obiectul** pentru a realiza legarea.

- **Static binding** este folosit frecvent la metodele supraîncărcate (**overloaded** methods), **Dynamic binding** (dynamic dispatch) este asociat în general cu metodele supradefinite (**overriding** methods).

Static Binding: Exemplu Java

```
public class StaticBindingTest {
    public static void main(String args[])  {
          Collection c = new HashSet();
          StaticBindingTest et = new StaticBindingTest();
          et.sort(c);
     }
  // metoda supraincarcata cu argument Collection
    public Collection sort(Collection c){
          System.out.println("In metoda sort(Collection)!");
          return c;
       }
   /*metoda supraincarcata cu argument HashSet, subclasa
    a lui Collection */
   public Collection sort(HashSet hs){
          System.out.println("In metoda sort(HashSet )!");
          return hs;
       }
    }
```

Output: In metoda sort(Collection)!

Dynamic Binding: Exemplu Java

```
class Vehicle {
    public void start() {
            System.out.println("In metoda start din Vehicle!");
    }
}
class Car extends Vehicle {
    public void start() {
            System.out.println ("In metoda start din Car!");
    }
}
public class DynamicBindingTest {
  public static void main(String args[]) {
     Vehicle vehicle = new Car(); //tip Vehicle, dar obiect Car
     vehicle.start();   //start din clasa Car - start() e supradef
    }
}
```

Output: In metoda start din Car!

Dynamic Binding

- Conceptul de overriding
- Car extends Vehicle - supradefineşte start()
- Apelul lui start() pentru un obiect Vehicle, apelează start() din subclasa Car deoarece obiectul referit de tipul Vehicle este un obiect Car
- Aceasta se petrece la execuție pentru că obiectul este creat doar la execuție -> Dynamic binding in Java.
- Dynamic binding este mai încet decât static binding pentru că apare în momentul execuției şi necesită timp pentru a determina care metodă este apelată de fapt.

Exemplu Static Binding vs Dynamic binding

```
public class Animal {
        public String type = "mamifer";
        public void show() {
            System.out.println(“Animalul este un: " + type);
        }
}
public class Dog extends Animal {
        public String type;
        public Dog(String type){
            this.type = type;
        }
        public void show() {
            System.out.println(“Cainele este un: " + type);
        }
}
```

```
…
Animal doggie = new Dog(“ciobanesc");
 doggie.show();
System.out.println (“Tipul este: " + doggie.type);
....
```

Exemplu Static Binding vs Dynamic binding

Output:

“Cainele este un: ciobanesc“ (dynamic binding)

“Tipul este: mamifer" (static binding)

Ce se afiseaza daca comentam:

public String type;

Observații

- Datele nu sunt niciodată supradefinite, doggie.type foloseşte Animal.type -- "static binding“

Avantaj: comportamentul este legat de tipul obiectului invocat, fără ca noi să ştim precis tipul acestuia.

Exemplu: Daca trimitem un Animal, nu ştim dacă este Cat/Dog/altceva, dar el va adopta comportamentul adecvat:

```
public void makeNoise(List<Animal> animals) {
      for (Animal a : animals) {
         a.makeNoise();
      }
}
```

- Fiecare animal din lista va produce propriul zgomot (va mieuna, lătra, etc)
- **Observație**: Clasa Animal poate fi abstracta. Astfel, comportamentul va fi definit de clasele concrete derivate din ea.

Exemplu

```
public class Shape {
    int x= 10;
    void draw() {
        System.out.println(“Shape”);
    }
}
public class Circle extends Shape{
    int x=5;
    void draw(){
        System.out.println(“Circle”);
    }
}
public class Test{
    public static void main(String[] args) {
        Shape shape = new Circle();
        shape.draw(); // DYNAMIC BINDING
        System.out.println(shape.x); // STATIC BINDING
    }
}
```

Explicații

- Atunci când compilatorul vede apelul draw(), el ştie că obiectul respectiv este de tipul Shape, dar el ştie şi că acel obiect poate fi o referință la orice clasă derivată din Shape.
- De aceea, compilatorul nu ştie ce versiune a metodei draw() este de fapt apelată. Aceasta se va şti doar în momentul execuției instrucțiunii respective:
- **shape.draw();** - metoda draw din Circle

- În unele cazuri, compilatorul poate determina versiunea corectă.
- În Java, variabilele membru prezintă static binding, deoarece Java nu permite comportament polimorfic pentru variabilele membru.
- Aceasta înseamnă că atât clasa Shape cât şi clasa Circle au o câte o variabilă membru cu acelaşi nume:

System.out.println(shape.x); - valoarea lui x din Shape.

Concluzie - dynamic binding

- metodele suprascrise în clasele derivate vor fi apelate folosind dynamic binding (late binding).
- este un mecanism prin care compilatorul, în momentul în care nu poate determina implementarea unei metode în avans, lasă la latitudinea JVM-ului (maşinii virtuale) alegerea implementării potrivite, în funcție de tipul real al obiectului.
- această legare a implementării de numele metodei la **momentul execuției** stă la baza polimorfismului.

Ce va afişa următorul program?

```
class TestEgal{
    public boolean equals ( TestEgal other ) {
        System.out.println( "In equals din TestEgal" );   return false;
    }
    public static void main( String [] args ) {
        Object t1 = new TestEgal(), t2 = new TestEgal();
        TestEgal t3 = new TestEgal();
        Object o1 = new Object();
        int count = 0;
        System.out.println( count++ ); // afiseaza 0
        t1.equals( t2 ) ;
        System.out.println( count++ ); // afiseaza 1
        t1.equals( t3 );
        System.out.println( count++ ); // afiseaza 2
        t3.equals( o1 );
        System.out.println( count++ ); // afiseaza 3
        t3.equals(t3);
        System.out.println( count++ ); // afiseaza 4
        t3.equals(t2);
    }
}
```

Rezolvare

Output:

0

1

2

3

In equals din TestEgal

4

Atenție! Este vorba de overload nu override!

Metoda *public boolean equals (TestEgal other)*

Nu supradefineşte metoda *public boolean equals (Object other)* din clasa Object, ci o supraîncarcă!

Deci, static binding! Singurul loc în care se apleează equals din TestEgal este **t3.equals(t3);** !

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!
   }
}
```

Clase abstracte şi Interfețe

Programare Orientată pe Obiecte

Clase şi metode abstracte

Clasă abstractă:

```
[public] abstract class ClasaAbstracta ... {
    // Declaratii uzuale
    // Declaratii de metode abstracte
}
```

Metodă abstractă: doar interfața, nu si implementarea

```
abstract class ClasaAbstracta {
    abstract void metodaAbstracta(); // Corect
    void metoda(); // Eroare
}
```

- **O metodă abstractă nu poate apărea decât într-o clasă abstractă!**
- **Orice clasă care are o metodă abstractă trebuie declarată ca fiind abstractă!**

Exemple:

Number: Integer, Double, …

Component: Button, List, ...

Clase abstracte

- **Clasă abstractă:** interfața comună, funcționalitate diferită pentru fiecare subtip, ce anume au clasele derivate în comun.
- **Creăm o clasă abstractă pentru:**
- ✓ **manipularea unui set de clase printr-o interfață comună**
- ✓ **reutilizarea unei serii de metode si membri din această clasă in clasele derivate.**
- Metodele suprascrise în clasele derivate vor fi apelate folosind dynamic binding (late binding)!
- O clasă abstractă poate să nu aibă nici o metodă abstractă!
- *Nu se pot crea instanțe ale unei clase abstracte, aceasta exprimând doar un punct de pornire pentru definirea unor instrumente reale!* => crearea unui obiect al unei clase abstracte eroare la compilare

Clase abstracte în contextul moştenirii

- O clasă care moşteneşte o clasă abstractă este ea însăşi abstractă daca nu implementează **toate** metodele abstracte ale clasei de bază.

⇒ O clasă care poate fi instanțiată (nu este abstractă) şi care moşteneşte o clasă abstractă trebuie să implementeze toate metodele abstracte pe lanțul moştenirii

- Este posibil să declarăm o **clasă abstractă fără** ca ea să aibă **metode abstracte** - când declarăm o clasă pentru care nu dorim instanțe (nu este corect *conceptual* să avem obiecte de tipul acelei clase, chiar dacă definiția ei este completă).

Interfețe

- Ce este o interfață ?
- Definirea unei interfețe
- Implementarea unei interfețe
- Interfețe şi clase abstracte
- Moştenire multiplă prin interfețe
- Utilitatea interfețelor
- Transmiterea metodelor ca parametri
- Compararea obiectelor
- Adaptori

Ce este o interfață ?

- **Colecție de metode abstracte şi declarații de constante**

- Defineşte un set de metode dar nu specifică nici o implementare pentru ele.
- Duce conceptul de clasă abstractă cu un pas înainte prin eliminarea oricăror implementări de metode
- Separarea modelului de implementare
- Protocol de comunicare
- O clasă care implementează o interfață trebuie obligatoriu să specifice implementări pentru toate metodele interfeței, supunându-se aşadar unui anumit comportament.
- Defineşte noi tipuri de date
- Clasele pot implementa interfețe

Definirea unei interfețe

```
[public] interface NumeInterfata
[extends SuperInterfata1, SuperInterfata2...]
{
    /* Corpul interfetei:
    Declarații de constante
    Declarații de metode abstracte
    */
}
```

Corpul unei interfețe poate conține:

- constante: acestea pot fi sau nu declarate cu modificatorii **public, static şi final** care sunt **impliciți**, nici un alt modificator neputând apărea în declarația unei variabile dintr-o interfață.
  - Constantele unei interfețe trebuie obligatoriu inițializate, însă pot fi inițializate cu **valori neconstante -** vor fi inițializate la inițializarea clasei.
- metode fără implementare: acestea pot fi sau nu declarate cu modificatorul **public**, care este **implicit**; nici un alt modificator nu poate apărea în declarația unei metode a unei interfețe.

Definirea unei interfețe

Atenție!

- Variabilele unei interfețe sunt implicit **publice** chiar dacă nu sunt declarate cu modificatorul public.
- Variabilele unei interfețe sunt implicit **constante** chiar dacă nu sunt declarate cu modificatorii static şi final.
- Metodele unei interfețe sunt implicit **publice** chiar dacă nu sunt declarate cu modificatorul public.

```
interface Exemplu {
    int MAX = 100;    // echivalent cu:
    public static final int MAX = 100;
    int MAX;   // Incorect, lipseste initializarea
    private int x = 1;  // Incorect, modificator nepermis
    void metoda();    // Echivalent cu:
    public void metoda();
    protected void metoda2();
    // Incorect, modificator nepermis
}
```

Exemplu initializare

- Expresia de inițializare este evaluată şi asignată exact o dată, atunci când interfața este inițializată.
- O eroare de compilare apare dacă o expresie de inițializare pentru un câmp conține o referință la un nume de câmp care este declarat mai târziu.
- Exemplu:

```
interfaceTest {
    float f=j;
    int j=1;
    int k=k+1;
}
```

Erori de compilare:

- **j** este referit la inițializarea lui **f**, înainte ca **j** să fie declarat
- inițializarea lui **k** se referă la însuşi **k**.

Implementarea unei interfețe

class NumeClasa implements NumeInterfata

sau:

class NumeClasa implements Interfata1, Interfata2, ...

- O clasă care implementează o interfață, pentru a fi instanțiabilă trebuie obligatoriu să specifice cod pentru toate metodele interfeței.
- O clasă poate avea şi alte metode şi variabile membre în afară de cele definite în interfață.
- Implementarea unei interfețe poate să fie şi o clasă abstractă.
- ***Spunem că un obiect are tipul X, unde X este o interfață, dacă acesta este o instanță a unei clase ce implementează interfața X.***
- Atenție! Modificarea unei interfețe implică modificarea tuturor claselor care implementează acea interfață.

Exemplu: implementarea unei stive (1)

- Interfața ce descrie stiva:

```
public interface Stack {
    void push ( Object item ) throws StackException ;
    void pop () throws StackException ;
    Object peek () throws StackException ;
    boolean empty ();
    String toString ();
}
```

- Clasa ce defineşte o excepție proprie StackException:

```
public class StackException extends Exception {
    public StackException () {
        super ();
    }
    public StackException ( String msg) {
        super (msg);
    }
}
```

Exemplu: implementarea unei stive folosind un vector

```
public class StackImpl1 implements Stack {
    private Object items [];   // Vect. ce contine ob.
    private int n=0;  // Nr. curent de elem. din stiva
    public StackImpl1 ( int max ) {   // Constructor
        items = new Object [ max ];
    }
    public StackImpl1 () {
        this (100) ;
    }
    public void push ( Object item ) throws
    StackException {
        if (n == items . length )
            throw new StackException (" Stiva e
    plina !");
        items [n++] = item ;
    }
```

Exemplu: implementarea unei stive folosind un vector (2)

```
    public void pop () throws StackException {
        if ( empty ())
            throw new StackException (" Stiva e vida !");
        items [--n] = null ;
    }
    public Object peek () throws StackException {
        if ( empty ())
            throw new StackException (" Stiva e vida !");
        return items [n -1];
    }
    public boolean empty () {
        return n ==0 ;
    }
    public String toString () {
        String s="";
        for (int i=n -1; i >=0; i --)
                s += items [i] + " ";
        return s;
    }
}
```

Exemplu: implementarea unei stive folosind o lista inlantuita (1)

```
public class StackImpl2 implements Stack {
    class Node {  // Clasa interna ce reprezinta un nod al listei
        Object item ; // informatia din nod
        Node link ; // legatura la urmatorul nod
        Node ( Object item , Node link ) {
            this . item = item ;
            this . link = link ;
        }
    }
    private Node top= null ; // Referinta la varful stivei
    public void push ( Object item ) {
        Node node = new Node (item , top);
        top = node ;
    }
    public void pop () throws StackException {
        if ( empty ())
          throw new StackException (" Stiva este vida !");
        top = top . link ;
    }
```

Exemplu: implementarea unei stive folosind o lista inlantuita (2)

```
    public Object peek () throws StackException {
        if ( empty ())
            throw new StackException (" Stiva este vida !");
        return top. item ;
    }
    public boolean empty () {
            return (top == null );
    }
    public String toString () {
            String s="";
            Node node = top;
            while ( node != null ) {
                        s +=  node . item  + " ";
                        node = node . link ;
            }
            return s;
    }
}
```

Observații

- Deşi metoda push din interfață declară aruncarea unor excepții de tipul StackException, nu este obligatoriu ca metoda din clasă să specifice şi ea acest lucru, atâta timp cât nu generează excepții de acel tip!
- Invers este însă obligatoriu!

Folosirea stivei:

```
public class TestStiva {
  public static void afiseaza ( Stack s) {
     System . out. println (" Continutul stivei este : " + s);
  }
  public static void main ( String args []){
          try {
              Stack s1 = new StackImpl1 ();
               s1. push ("a");
               s1. push ("b");
               afiseaza (s1);
               Stack s2 = new StackImpl2 ();
               s2. push ( new Integer (1));
               s2. push ( new Double (3.14) );
               afiseaza (s2);
          } catch ( StackException e) {
              System . err. println (" Eroare la lucrul cu stiva!");
              e. printStackTrace ();
        }
    }
}
```

Interfețe şi clase abstracte

- “O clasă abstractă nu ar putea înlocui o interfață ?”
- Unele clase sunt forțate să extindă o anumită clasă (de exemplu orice applet trebuie să fie subclasa a clasei Applet) şi nu ar mai putea sa extindă o altă clasă. Fără folosirea interfețelor nu am putea forța clasa respectivă să respecte diverse tipuri de protocoale
- Extinderea unei clase abstracte forțează o relație între clase
- Implementarea unei interfețe specifică doar necesitatea implementării unor anumite metode
- Interfețele şi clasele abstracte nu se exclud, fiind folosite “împreună”:
    - •List – interfață
    - •AbstractList – clasă abstractă, implementează interfața List
    - •LinkedList, ArrayList – clase concrete, instanțiabile derivate din AbstractList!

Moştenire multiplă prin interfețe

- class NumeClasa extends ClasaUnica implements Interfata1, Interfata2, ...
- interface NumeInterfata extends Interfata1, Interfata2, ...
- Ierarhia interfețelor este independentă de ierarhia claselor care le implementează.

```
interface I1 {
    int a=1;
    void metoda1();
}
interface I2 {
    int b=2;
    void metoda2();
}
class C implements I1, I2 {
    public void metoda1() {...}
    public void metoda2() {...}
}
```

Ambiguități

```
interface I1 {
    int x=1;
    void metoda();
}
interface I2 {
    int x=2;
    void metoda(); //corect
    //int metoda(); //incorect
}
class C implements I1, I2 {
    public void metoda() {
            System.out.println(I1.x); //corect
            System.out.println(I2.x); //corect
            System.out.println(x); //ambiguitate
    }
}
```

Utilitatea interfețelor

- Definirea unor similarități între clase independente.
- Impunerea unor specificații: asigură că toate clasele care implementează o interfață pun la dispoziție metodele specificate în interfață - de aici rezultă posibilitatea implementării unor clase prin mai multe modalități şi folosirea lor într-o manieră unitară;
- Definirea unor grupuri de constante
- Transmiterea metodelor ca parametri

- Crearea grupurilor de constante:

```
public interface Luni {
    int IAN=1, FEB=2, ..., DEC=12;
}
...
if (luna < Luni.DEC)
    luna ++;
else
    luna = Luni.IAN;
```

Transmiterea metodelor ca parametri

```
interface Functie {
    void executa(Nod u);
}
class Graf {
    void explorare(Functie f) {
            ...
            if (explorarea a ajuns in nodul v)  f.executa(v);
    }
}
//Definim diverse functii
class AfisareRo implements Functie {
    public void executa(Nod v) {
            System.out.println("Nodul curent este: " + v);
    }
}
class AfisareEn implements Functie {
    public void executa(Nod v) {
            System.out.println("Current node is: " + v);
    }
}
```

Transmiterea metodelor ca parametri (2)

```
public class TestCallBack {
    public static void main(String args[]) {
        Graf G = new Graf();
        Functie f1 = new AfisareRo();
        G.explorare(f1);
        Functie f2 = new AfisareEn();
        G.explorare(f2);
    /* sau mai simplu:
        G.explorare(new AfisareRo());
        G.explorare(new AfisareEn());
    */
    }
}
```

Interfața FilenameFilter

- folosită pentru a crea filtre pentru fişiere
- sunt primite ca argumente de metode care listează conținutul unui director, cum ar fi metoda *list* a clasei File.
- putem spune că metoda *list* primeşte ca argument o altă funcție care specifică dacă un fişier va fi returnat sau nu (criteriul de filtrare).
- Exemplu: Listarea fişierelor din directorul curent care au anumită extensie primită ca argument. Dacă nu se primeşte nici un argument , vor fi listate toate.

```
import java .io .*;
class Listare {
   public static void main ( String [] args ) {
        try {
                File director = new File (".");
                String [] list ;
```

Interfața FilenameFilter: exemplu

```
    if ( args . length > 0)
        list = director . list ( new Filtru ( args [0]) );
    else
        list = director . list ();
    for (int i = 0; i < list . length ; i ++)
        System . out. println ( list [i]);
    } catch ( Exception e) {   e. printStackTrace (); }
  }
}
class Filtru implements FilenameFilter {
    String extensie ;
    Filtru ( String extensie ) {
        this . extensie = extensie ;
    }
    public boolean accept ( File dir , String nume ) {
        return ( nume . endsWith ("." + extensie ) );
    }
}
```

Compararea obiectelor

Exemplu: Clasa Persoana (fără suport pentru comparare)

```
class Persoana {
    private int cod ;
    private String nume ;
    public Persoana ( int cod , String nume ) {
        this .cod = cod;
        this . nume = nume ;
    }
    public String toString () {
        return cod + " \t " + nume ;
    }
}
```

Exemplu: Sortarea unui vector de tip referință

```
class Sortare {
    public static void main ( String args []) {
        Persoana p[] = new Persoana [3];
        p[0] = new Persoana (3, " Ionescu ");
        p[1] = new Persoana (1, " Vasilescu ");
        p[2] = new Persoana (2, " Georgescu ");
        java . util . Arrays . sort (p);
        System . out. println (" Persoanele ordonate dupa cod:");
        for (int i=0; i<p. length ; i++) System . out. println (p[i]);
    }
}
```

Interfața Comparable

- Interfața Comparable impune o ordine totală asupra obiectelor unei clase ce o implementează. Această ordine se numeşte ordinea naturală a clasei şi este specificată prin intermediul metodei compareTo. Definiția interfeței este:

```
public interface Comparable {
    int compareTo(Object o);
}
```

- metoda compareTo trebuie să returneze:

• o valoare strict negativă: dacă obiectul curent (this) este mai mic decât obiectul primit ca argument;

• zero: dacă obiectul curent este egal cu obiectul primit ca argument;

• o valoare strict pozitivă: dacă obiectul curent este mai mare decât obiectul primit ca argument.

Interfața Comparable. Exemplu

Exemplu: Clasa Persoana cu suport pentru comparare

```
class Persoana implements Comparable {
    private int cod ;
    private String nume ;
    public Persoana ( int cod , String nume ) {
        this .cod = cod;
        this . nume = nume ;
    }
    public String toString () {
        return cod + " \t " + nume ;
    }
    public boolean equals ( Object o) {
        if (!( o instanceof Persoana ))  return false ;
        Persoana p = ( Persoana ) o;
        return (cod == p.cod) && ( nume . equals (p. nume ));
    }
    public int compareTo ( Object o) {
        if (o== null ) throw new NullPointerException ();
        if (!( o instanceof Persoana ))
           throw new ClassCastException ("Nu pot compara !");
        Persoana p = ( Persoana ) o;
        return (cod - p.cod);
    }
}
```

Interfața Comparator

- In cazul în care dorim să sortăm elementele unui vector ce conține referințe după alt criteriu decât ordinea naturală a elementelor
- Interfața java.util.Comparator conține metoda compare, care impune o ordine totală asupra elementelor unei colecții.

  int compare(Object o1, Object o2);

```
class MyComp implements Comparator{
  public int compare ( Object o1 , Object o2) {
    Persoana p1 = ( Persoana )o1;
    Persoana p2 = ( Persoana )o2;
    return (p1. nume . compareTo (p2. nume ));
}
…
Arrays.sort(p, new MyComp());
```

Interfața Comparator

Exemplu: Sortarea unui vector folosind un comparator

```
import java . util .*;
class Sortare {
    public static void main ( String args []) {
            Persoana p[] = new Persoana [4];
            p[0] = new Persoana (3, " Ionescu ");
            p[1] = new Persoana (1, " Vasilescu ");
            p[2] = new Persoana (2, " Georgescu ");
            p[3] = new Persoana (4, " Popescu ");
            Arrays . sort (p, new Comparator () {
                        public int compare ( Object o1 , Object o2) {
                            Persoana p1 = ( Persoana )o1;
                            Persoana p2 = ( Persoana )o2;
                            return (p1. nume . compareTo (p2. nume ));
                        }
            });
            System . out. println (" Persoanele ordonate dupa nume :");
            for (int i=0; i<p. length ; i++)
                        System . out. println (p[i]);
    }
}
```

Problema

- Să se definească o clasă SortedVector derivată din Vector, care să permită ordonarea după orice criteriu, specificat de utilizator la construirea unui obiect SortedVector. Clasa va conține o variabilă de tip Comparator, inițializată de un constructor cu argument de tip Comparator şi folosită de metoda Collections.sort.
- Să se definească o clasă Pair care conține două variabile de tip Object, cu metodele equals şi toString redefinite.
- Să se scrie un program pentru crearea a doi vectori SortedVector de obiecte Pair, unul ordonat după primul obiect din pereche şi celălalt ordonat după al doilea obiect.

  OBS:Clasa Pair conține o variabilă String şi o variabilă Integer.