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

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
}
```

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 ) );
    }
}
```

Clase incluse (interne, imbricate, *nested classes*)

- **Clase declarate în interiorul unei alte clase**
- Reprezintă o funcționalitate importantă
  - permit gruparea claselor care sunt legate logic
  - controlul vizibilității uneia din cadrul celorlalte.
- Se comportă ca un **membru** al clasei => o clasă internă are acces la toți membrii clasei de care aparține (*outer class*), inclusiv cei private!

Mai multe tipuri, în funcție de modul de a le instanția şi de relația lor cu clasa exterioră:

1. clase **interne** normale (*regular inner classes*) - **membru**
2. clase **interne metodelor** (*method-local inner classes*) sau blocurilor **- locale**
3. clase **anonime** (*anonymous inner classes*)
4. clase incluse statice (*static nested classes*)

Clase interne

- o clasă membră a unei alte clase, numită şi clasă de acoperire.

```
class ClasaDeAcoperire{
    class ClasaImbricata1 {
        // Clasa membru
        // Acces la membrii clasei de acoperire
    }
    void metoda() {
        class ClasaImbricata2 {
            // Clasa locala metodei
            // Acces la mebrii clasei de acoperire si
            // la variabilele finale ale metodei
        }
    }
}
```

- **Identificare claselor imbricate**
  - **ClasaDeAcoperire.class**
  - **ClasaDeAcoperire$ClasaImbricata1.class**
  - **ClasaDeAcoperire$ClasaImbricata2.class**

Clase interne

```
class ClasaDeAcoperire{
    private int x=1;
    class ClasaImbricata1 {
        int a=x;
    }
    void metoda() {
        final int y=2;
        int z=3;
        class ClasaImbricata2 {
          int b=x;
          int c=y;
          int d=z; // Incorect
        }
    }
}
```

Modificatori de acces - clase interne (*nested classes*)

- Clasele membru (1) pot fi declarate cu modificatorii **public, protected, private** sau **implicit** pentru a controla nivelul lor de acces din exterior.

- Pentru clasele imbricate locale unei metode(2) nu sunt permişi aceşti modificatori!

- Toate clasele imbricate pot fi declarate folosind modificatorii **abstract** şi **final**.

- **Clasa care conține alte clase poate avea doar modificatorul public şi cel implicit!**

Clase interne membru (1)

- Compilatorul creează fişiere *.class* separate pentru fiecare clasă internă
- Clasa internă poate fi referită din exteriorul clasei de acoperire folosind expresia
  ClasaExternă.ClasaInternă
- nu este permisă execuția fişierului
  ClasaExternă$ClasaInternă.class
- Dintr-o clasă internă putem accesa **referința la clasa externă**:
  numeClasăExternă.this
- Două modalități de a obține o instanță a clasei interne:

➢ definim o metodă (***getInnerInstance)*** care creează şi întoarce o astfel de instanță;

➢ instanțiem efectiv clasa internă;

- Pentru a instanția clasa internă avem nevoie de o instanta a clasei externe

Clase interne membru (1) - Exemplu

```
class Outer {
      class Inner {
            private int i;
            public Inner (int i) {  this.i = i; }
            public int value () {  return i; }
      }
      public Inner getInnerInstance () {
            Inner in = new Inner (11);
            return in; }
}
public class Test {
      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());
      }
}
```

Problemă

- Ce se petrece dacă declarăm clasa Inner cu modificatorul private?

```
class Outer {
    private class HiddenInner {
        private int i;
        public HiddenInner  (int i) {  this.i = i; }
        public int value () {  return i; }
    }
    public HiddenInner getInnerInstance () {
        HiddenInner in = new Inner (11);
        return in; }
}
```

- În acest mod, vizibilitatea ei a fost redusă pentru că nu poate fi instanțiată decât în această funcție!

- O putem accesa din exteriorul clasei Outer?

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

Clase interne membru (1) – Exemplu clasă ascunsă

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

class Outer {
    private class HiddenInner implements Hidden {
        private int i;
        public HiddenInner (int i) {  this.i = i; }
        public int value () { return i; }
    }
    public Hidden getInnerInstance () {
        HiddenInner in = new HiddenInner(11);
        return in;
    }
}
```

Clase interne membru (1) – Exemplu clasă ascunsă

```
public class Test {
  public static void main(String[] args) {
    Outer out = new Outer();

    Outer.HiddenInner in1 = out.getInnerInstance();
    // eroare, tipul Outer.HiddenInner nu este vizibil!

    Outer.HiddenInner in2 = new Outer().new
    HiddenInner(10);  // din nou eroare

    Hidden in3= out.getInnerInstance();
     // acces corect la o instanta HiddenInner

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

Clase interne în metode - locale

```
interface Hidden {
    public int value ();}
class Outer {
    public Hidden getInnerInstance() {
        class FuncInner implements Hidden {
         private int i = 11;
         public int value () {return i;}
         }
         return new FuncInner();
    }
}
public class Test {
    public static void main(String[] args) {
       Outer out  = new Outer ();
       Outer.FuncInner in2 = out.getInnerInstance();
            // EROARE: clasa FuncInner nu este vizibila
        Hidden in3 = out.getInnerInstance();
       System.out.println(in3.value());
    }
}
```

Clase interne în metode - locale

- Singurii modificatori care pot fi aplicați acestor clase sunt abstract sau final
- nu pot folosi variabilele declarate în metoda respectivă şi nici parametrii metodei
- Pentru a le putea accesa, variabilele trebuie declarate **final**
- Explicație:
- Variabilele şi parametrii metodelor se află pe segmentul de stivă (zonă de memorie) creat pentru metoda respectivă, ceea ce face ca ele să nu existe la fel de mult cât clasa internă.
- Dacă variabila este declarată final, atunci la runtime se va stoca o copie a acesteia ca un câmp al clasei interne, în acest mod putând fi accesată şi după execuția metodei.

Clase interne în metode - Exemplu

```
public void f() {
    final Student s = new Student();
    /* s tb declarat final ca sa poata fi accesat
    din ModStudent */

    class ModStudent {
        public void modData() {
            s.name = ...                // OK
            s = new Student();      // GRESIT!
        }
    }
}
```

Clase interne în blocuri - Exemplu

```
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;
    }
}
```

Clase interne în blocuri

Observații:

- Definirea clasei interne în cadrul unui bloc *if* nu înseamnă că declarația va fi luată în considerare doar la rulare, în cazul în care condiția este adevarată!
- Semnificația declarării clasei într-un bloc este legată strict de vizibilitatea acesteia!
- La compilare, clasa va fi creată indiferent care este valoarea de adevăr a condiției *if!*

Clase anonime

Clasa anonimă = clasă internă locală fără nume folosită pentru instanțierea unui singur obiect.

- sunt foarte utile în crearea unor obiecte ce implementează o anumită interfață sau extind o anumită clasă abstractă.

```
metoda(new Interfata() {
  // Implementarea metodelor interfetei
});
```

Exemplu:

```
if (args.length > 0) {
    final String extensie = args[0];
    list = director.list ( new FilenameFilter() {
        /*Clasă internă anonimă ⇔ creează un obiect al unei
    clase anonime ce implementeaza FilenameFilter! */
        public boolean accept (File dir, String nume) {
          return ( nume.endsWith("." + extensie) );
        }
    } );
}
```

Clase anonime

Pot extinde o clasă *sau* să implementeze o singură interfață:

- nu pot face ambele ca la clasele ne-anonime (interne sau nu)
- nici nu pot să implementeze mai multe interfețe.

Nu pot avea **constructori!**

- Clasa este creată cu constructorul *implicit!*
- Dacă dorim să invocăm un **alt constructor** al clasei de bază -> transmiterea parametrilor către constructorul clasei de bază **direct** la crearea obiectului de tip clasă anonimă:

new Student("Mihai") { ... }

- am instanțiat o clasa anonimă, ce extinde clasa Student, apelând constructorul clasei de bază cu parametrul "*Mihai*".

Utilizarea claselor interne

- Pentru o clasă care:
  - să nu fie **accesibilă** din exterior sau
  - **nu** mai are **utilitate** în alte zone ale programului
- Implementăm o anumită interfață şi vrem să întoarcem o referință la acea interfață, **ascunzând** în acelaşi timp implementarea.
- Dorim să folosim/extindem funcționalități ale mai **multor** clase -> Putem defini clase interioare. Acestea pot **moşteni** orice clasă şi au, în plus, acces la clasa **exterioară**.
- Implementarea unei arhitecturi de control, marcată de nevoia de a trata evenimente într-un **sistem bazat pe evenimente**.
- Exemplu: Swing - **GUI** (graphical user interface)

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]);
    }
}
```

Adaptori

- In cazul în care o interfață conține mai multe metode şi, la un moment dat, avem nevoie de un obiect care implementează interfața respectivă dar nu specifică cod decât pentru o singură metodă, el trebuie totuşi să implementeze toate metodele interfeței, chiar dacă nu specifică nici un cod.

```
interface X {
        void metoda_1();
        void metoda_2();
        ...
        void metoda_n();
}

class test implements X {
        public void metoda_1() {
                // Singura metoda care ne intereseaza
                ...
        }
        // Trebuie sa apara si celelalte metode chiar daca nu
        //au implementare efectiva
```

Adaptori

```
    public void metoda_2() {}
    public void metoda_3() {}
    ...
    public void metoda_n() {}
});
```

- Un adaptor este o clasă abstractă care implementează o anumită interfață fără a specifica cod nici unei metode a interfeței.

```
public abstract class XAdapter implements X {
    public void metoda_1() {}
    public void metoda_2() {}
    ...
    public void metoda_n() {}
}
functie(new XAdapter() {
    public void metoda_1() {
    // Singura metoda care ne intereseaza
            ...
    }
});
```

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.