Interfețe

Programare Orientată pe Obiecte

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.
- 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 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]. toString () + " ";
        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 ). toString () + " ";
                        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

•AbstractList

•LinkedList, ArrayList

Moştenire multiplă prin interfețe

class NumeClasa 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 similaritati 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

- folosite 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 ) );
    }
}
```

Folosirea claselor anonime

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

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

Exemplu:

```
if (args.length > 0) {
    final String extensie = args[0];
    list = director.list ( new FilenameFilter() {
                    // Clasă internă anonimă
          public boolean accept (File dir, String nume) {
            return ( nume.endsWith("." + extensie) );
          }
    } );
}
```

Compararea obiectelor

Exemplu: Clasa Persoana (fără suport pentru comparare)

```
class Persoana {
    int cod ;
    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 {
    int cod ;
    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
            ...
    }
});
```