



Programare avansată

Interfețe

Ce este o interfață ?

DEX: (inform.) frontieră convențională între două sisteme sau unități, care permite schimburi de informații după anumite reguli.



Contract, protocol de comunicare

- Interfețele descriu un model, contract
- Clasele implementează modelul, aderă la contract



Definirea unei interfete

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

Exemple:

```
public interface Student {
    int NOTA_MAXIMA = 5; -----> //public static final int ...
    int getNotaExamen(); -----> //public abstract int ...
}

public interface AutoCloseable {
    void close();
}
```

Implementarea unei interfețe

```
class NumeClasa implements Interfata1, Interfata2, ... {  
    /* O clasă concretă care implementează o interfață  
       trebuie obligatoriu să specifice cod pentru  
       toate metodele interfeței */  
}
```

Exemple:

```
public class StudentInformatica implements Student {  
    public int getNotaExamen {  
        return NOTA_MAXIMA;  
    }  
}
```

```
public class FileReader implements AutoCloseable { ... }
```

```
public class Connection implements AutoCloseable { ... }
```

Interfața – Tip referință

Spunem că un obiect este de tipul X, unde X este o interfață, dacă acesta este o instanță a unei clase ce implementează interfața X.

```
Student student = new StudentInformatica();
```

```
Student student = new Student();
```

```
AutoCloseable reader = new FileReader("fis.txt");
```

```
AutoCloseable reader = new AutoCloseable();
```

Interfață – Implementări multiple

```
public interface Matrix {  
    void set(int row, int col, double value);  
    double get(int row, int col);  
    Matrix add(Matrix m);  
    Matrix mul(Matrix m);  
    ...  
}  
public class DefaultMatrixImpl implements Matrix {  
    private double[][] data;  
    public DefaultMatrixImpl(int rows, int cols) { ... }  
    ...  
}  
public class SparseMatrixImpl implements Matrix {  
    private int[] row;  
    private int[] col;  
    private double[] data;  
    public SparseMatrixImpl(int rows, int cols) { ... }  
    ...  
}  
public static void main ( String args []){  
    Matrix a = new DefaultMatrixImpl(10, 10); a.set(0,0, 123); ...  
    Matrix b = new SparseMatrixImpl (10, 10); b.set(9,9, 456); ...  
    Matrix c = a.add(b);  
}
```

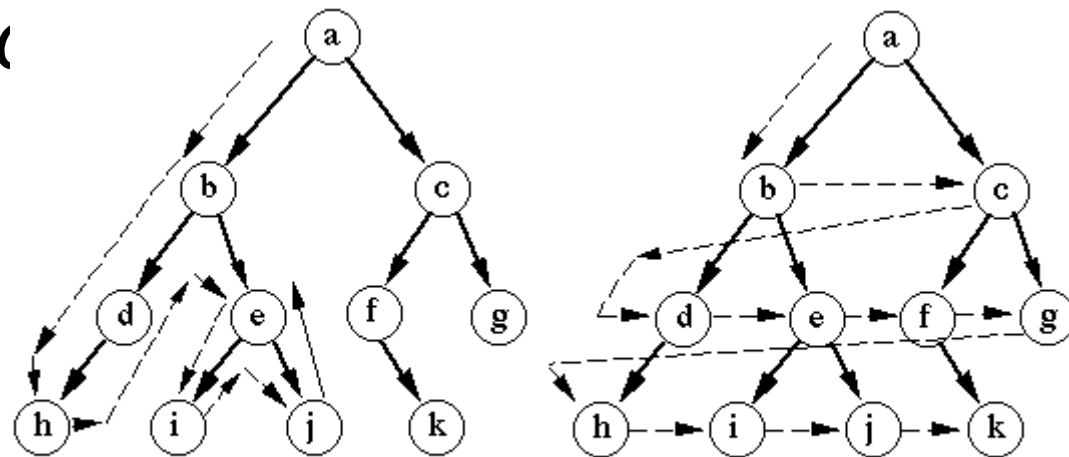
Interfețe și clase abstracte

- Extinderea unei clase abstracte forțează o relație între clase
- Implementarea unei interfete specifică doar aderarea la un anumit contract prin implementarea unor anumite metode
- Interfețele și clasele abstracte nu se exclud, fiind folosite împreună în multe situații:
 - ♦ **List**
 - ♦ **AbstractList**
 - ♦ **LinkedList, ArrayList**

Metode de tip *callback*

Cum putem transmite o anumită metodă (secvență de cod, funcționalitate) ca argument unei alte metode?

Exemplu: *Explorarea unui graf – când explorarea a ajuns într-un anumit nod, dorim executarea unei secvențe de cod*



Depth-first search

Breadth-first search

Implementarea metodelor *callback*

```
public interface Functie {
    public void executa(Nod u);
}

public class Graf {
    ...
    public 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("The current node is: " + v);
    }
}
```

```
Graf g = new Graf();
...
G.explorare(new AfisareRo());
G.explorare(new AfisareEn());
```

Interfața *FilenameFilter*

```
//Listarea fisierelor din directorul curent care au anumita extensie
import java.io.*;
```

```
public class Listare {

    public static void main ( String [] args ) {
        File director = new File (".");
        String[] list = director.list(new Filtru (args[0]));
        for (int i = 0; i < list.length ; i ++) {
            System.out.println(list[i]);
        }
    }
}
```

```
class Filtru implements FilenameFilter {
    String extensie ;
    public Filtru ( String extensie ) {
        this.extensie = extensie ;
    }
    public boolean accept ( File dir , String nume ) {
        return ( nume . endsWith ( "." + extensie ) );
    }
}
```

Clase anonime

Clasă anonimă = Clasă internă folosită pentru instanțierea unui singur obiect de un anumit tip.

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

Exemplu:

```
director.list(new FilenameFilter() {  
    // Clasa interna anonima  
    public boolean accept (File dir, String nume) {  
        return ( nume.endsWith(".") + extensie) );  
    }  
});
```

Compilare: ClasaDeAcoperire\$1.class, ClasaDeAcoperire\$2.class, ...



Compararea obiectelor

```
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 ;
    }
}
```

```
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 ");
        java.util.Arrays.sort(p);
        System.out.println ("Persoanele ordonate:");
        for (int i=0; i<p.length ; i++)
            System.out.println (p[i]);
    }
}
```



Interfața *Comparable*

Definește relația de ordine naturală a obiectelor unei clase

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

```
class Persoana implements Comparable {  
    ...  
  
    public int compareTo (Object other) {  
        //returneaza 0 this==other, <0 this<other, >0 this>other  
  
        if (other == null )  
            throw new NullPointerException();  
        if (!( other instanceof Persoana ))  
            throw new ClassCastException ("Nu pot compara !");  
  
        Persoana pers = (Persoana) other;  
        return (this.cod - pers.cod);  
    }  
}
```

Interfața *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]);
    }
}
```

Interfețe *marker*

- Sunt interfețe care nu definesc nicio metodă

```
interface Serializable {}
```

```
interface Cloneable {}
```

- Rolul lor este de a asocia *metadata* unei clase, care să fie folosite la execuție într-un anumit scop

```
class Persoana
```

```
    implements Serializable, Cloneable { ... }
```

- Alternativă: folosirea *adnotărilor*

```
@Entity(table="persoane")
```

```
class Persoana { ... }
```


Concluzii – Utilitatea interfețelor

- Impunerea unor specificații
- Definirea unui protocol de comunicare
- Separarea modelului de implementare
- Definirea unor similarități între clase independente pentru a le *referi* unitar
- Tip referință
- Implementarea metodelor callback
- Flexibilitate în proiectarea și scrierea codului