Curs 4
Interfete

Ce este o interfata ?

Interfetele duc conceptul de clasa abstracta cu un pas înainte prin eliminarea oricarei implementari a metodelor, punând în practica unul din conceptele POO de separare a modelului unui obiect (interfata) de implementarea sa. Asadar, o interfata poate fi privita ca un protocol de comunicare între obiecte.
O interfata Java defineste un set de metode dar nu specifica nici o implementare pentru ele. O clasa care implementeaza o interfata trebuie obligatoriu sa specifice implementari pentru toate metodele interfetei, supunându-se asadar unui anumit comportament.

Definitie

O *interfata* este o colectie de metode fara implementare si declaratii de constante

Definirea unei interfete

Definirea unei interfete se face prin intermediul cuvântului cheie **`interface`**:

```
[public] interface NumeInterfata
        [extends SuperInterfata1 [,extends SuperInterfata2...]]
{
        //corpul interfetei:constane si metode abstracte
}
```

O interfata poate avea un singur modificator: `public`. O interfata publica este accesibila tuturor claselor indiferent de pachetul din care fac parte. O interfata care nu este publica este accesibila doar claselor din pachetul din care face parte interfata.
O clasa poate extinde oricâte interfete. Acestea se numesc *superinterfete* si sunt separate prin virgula (vezi "Mostenirea multipla prin intermediul interfetelor").
Corpul unei interfete contine:

- *constante*: acestea pot fi sau nu declarate cu modificatorii `public, static` si `final` care sunt impliciti; nici un alt modificator nu poate aparea în declaratia unei variabile a unei interfete Constantele dintr-o interfata trebuie obligatoriu initializate.

```
interface NumeInterfata {
        int MAX = 100;                  //echivalent cu
        public static final MAX = 100;
        int MAX;                        //ilegal - fara initializare
```

```
        private int x = 1;              //ilegal
    }
```

- *metode fara implementare*: acestea pot fi sau nu declarate cu modificatorul `public` care este implicit; nici un alt modificator nu poate aparea în declaratia unei metode a unei interfete.

```
    interface NumeInterfata {
        void metoda();                  //echivalent cu
        public void metoda();
        protected void metoda2();       //ilegal
```

Atentie
Variabilele unei interfete sunt implicit publice chiar daca nu sunt declarate cu modificatorul `public`.
Variabilele unei interfete sunt implicit constante chiar daca nu sunt declarate cu modificatorii `static` si `final`.
Metodele unei interfete sunt implicit publice chiar daca nu sunt declarate cu modificatorul `public`.
In variantele mai vechi de Java era permis si modificatorul `abstract` în declaratia interfetei si în declaratia metodelor, însa a fost eliminat deoarece atât interfata cât si metodele sale sunt implicit abstracte.

Implementarea unei interfete

Se face prin intermediul cuvântului cheie **`implements`**:

```
class NumeClasa implements NumeInterfata sau
class NumeClasa implements Interfata1, Interfata2...
```

O clasa poate implementa oricâte interfete. (vezi "Mostenirea multipla prin intermediul interfetelor").
O clasa care implementeaza o interfata trebuie obligatoriu sa specifice cod pentru toate metodele interfetei. Din acest motiv, odata creata si folosita la implementarea unor clase, o interfata nu mai trebuie modificata , în sensul ca adaugarea unor metode noi sau schimbarea signaturii metodelor existente va duce la erori în compilarea claselor care o implementeaza.
Modificarea unei interfete implica modificarea tuturor claselor care implementeaza acea interfata!
Implementarea unei interfete poate sa fie si o clasa abstracta.

Exemplu de interfata

```
interface Instrument {
        //defineste o metoda fara implementare
        void play();
}

class Pian implements Instrument {
        //clasa care implementeaza interfata
        //trebuie obligatoriu sa implementeze metoda play
        public void play() {
                System.out.println("Pian.play()");
        }
}

class Vioara implements Instrument {
        //clasa care implementeaza interfata
```

```
        //trebuie obligatoriu sa implementeze metoda play
        public void play() {
                System.out.println("Vioara.play()");
        }
}

public class Muzica { //clasa principala
        static void play(Instrument i) {
                //metoda statica care porneste un instrument generic
                //ce implementeaza interfata Instrument
                i.play();
        }
        static void playAll(Instrument[] e) {
                for(int i = 0; i < e.length; i++)
                        play(e[i]);
        }
        public static void main(String[] args) {
                Instrument[] orchestra = new Instrument[2];
                int i = 0;
                orchestra[i++] = new Pian();
                orchestra[i++] = new Vioara();
                playAll(orchestra);
        }
}
```

Se observa ca folosind interfata `Instrument` putem adauga noi clase de instrumente fara a schimba codul metodelor `play` si `playAll` din clasa principala întrucât acestea primesc ca parametru un instrument generic.

Atentie
O interfata nu este o clasa, dar orice referinta la un obiect de tip interfata poate primi ca valoare o referinta la un obiect al unei clase ce implementeaza interfata respectiva (upcast). Din acest motiv interfetele pot fi privite ca tipuri de date.

Diferente între o interfata si o clasa abstracta

La prima vedere o interfata nu este altceva decât o clasa abstacta în care toate metodele sunt abstracte (nu au nici o implementare). Asadar o clasa abstracta nu ar putea înlocui o interfata ?
Raspunsul la intrebare este Nu. Deosebirea consta în faptul ca unele clase sunt fortate sa extinda o anumita clasa (de exemplu orice applet trebuie sa fie subclasa a clasei Applet) si nu ar mai putea sa extinda o clasa abstracta deoarece în Java nu exista decât mostenire simpla. Fara folosirea interfetelor nu am putea forta clasa respectiva sa respecte un anumit protocol.
La nivel conceptual diferenta consta în:

- extinderea unei clase abstracte forteaza o relatie între clase
- implementarea unei interfete specifica doar necesitatea implementarii unor anumie metode

Mostenire multipla prin intermediul interfetelor

Curs 4

Interfetele nu au nici o implementare si nu ocupa spatiu de memorie la instantierea lor. Din acest motiv nu reprezinta nici o problema ca anumite clase sa implementeze mai multe interfete sau ca o interfata sa extinda mai multe interfete (sa aiba mai multe superinterfete)

```
class NumeClasa implements Interfata1, Interfata2, ...
interface NumeInterfata extends Interfata1, Interfata2, ...
```

O interfata mosteneste atât constantele cât si declaratiile de metode de la superinterfetele sale. O clasa mosteneste doar constantele unei interfete.
Exemplu de clasa care implementeaza mai multe interfete:

```
interface Inotator {
        void inoata();
}
interface Zburator {
        void zboara();
}
class Luptator {
        public void lupta() {}
}
class Erou extends Luptator implements Inotator, Zburator {
        public void inoata() {}
        public void zboara() {}
}
```

Exemplu de interfata care extinde mai multe interfete :

```
interface Monstru {
        void ameninta();
}
interface MonstruPericulos extends Monstru {
        void distruge();
}
interface Mortal {
        void omoara();
}
interface Vampir extends MonstruPericulos, Mortal {
        void beaSange();
}
class Dracula implements Vampir {
        public void ameninta() {}
        public void distruge() {}
        public void omoara();
        public void beaSange() {}
}
```

Atentie
O clasa nu poate avea decât o superclasa
O clasa poate implementa oricâte interfete
O clasa mosteneste doar constantele unei interfete
O clasa nu poate mosteni implementari de metode dintr-o interfata
Ierarhia interfetelor este independenta de ierarhia claselor care le implementeaza

Utilitatea interfetelor

Curs 4

O interfata defineste un protocol ce poate fi implementat de orice clasa, indiferent de ierarhia de clase din care face parte. Interfetele sunt utile pentru:

- definirea unor similaritati între clase independente fara a forta artificial o legatura între ele.
- asigura ca toate clasele care implementeaza o interfata pun la dipozitie metodele specificate în interfata; de aici rezulta posibilitatea implementarii unitare a unor clase prin mai multe modalitati.
- specificarea metodelor unui obiect fara a deconspira implementarea lor (aceste obiecte se numesc *anonime* si sunt folosite la livrarea unor pachete cu clase catre alti programatori: acestia pot folosi clasele respective dar nu pot vedea implementarile lor efective)
- definirea unor grupuri de constante
- transmiterea metodelor ca parametri (tehnica Call-Back) (vezi "Transmiterea metodelor ca parametri").

Crearea grupurilor de constante

Deoarece orice variabila a unei interfete este implicit declarata cu `public, static` si `final` interfetele reprezinta o metoda convenabila de creare a unor grupuri de constante, similar cu `enum` din C++.

```
public interface Luni {
        int IAN=1, FEB=2, ..., DEC=12;
}
```

Folosirea acestor constante se face prin expresii de genul `NumeInterfata.constanta` :

```
if (luna < Luni.DEC)
        luna ++
else
        luna = Luni.IAN;
```

Transmiterea metodelor ca parametri (call-back)

Transmiterea metodelor ca parametri se face în C++ cu ajutorul pointerilor. In Java aceasta tehnica este implementata prin intermediul interfetelor. Vom ilustra acest lucru prin intermediul unui exemplu.

Explorarea unui graf
In fiecare nod trebuie sa se execute prelucrarea informatiei din el prin intermediul unei functii primite ca parametru.

```
interface functie {
        public int executie(int arg);
}

class Graf {
        //...
        void explorare(functie f) {
                //...
                if explorarea a ajuns in nodul v
                        f.executie(v.valoare);
```

```
                //...
        }
}

//Definim doua functii
class f1 implements functie {
        public int executie(int arg) {
                return arg+1;
        }
}
class f2 implements functie {
        public int executie(int arg) {
                return arg*arg;
        }
}

public class TestCallBack {
        public static void main(String args[]) {
                Graf G = new Graf();
                G.explorare(new f1());
                G.explorare(new f2());
        }
}
```

Interfata `FilenameFilter`

Instantele claselor ce implementeaza aceasta interfata sunt folosite pentru a crea filtre pentru fisiere si sunt primite ca argumente de metode care listeaza continutul unui director, cum ar fi metoda `list` a clasei `File`.
Aceasta interfata are o singura metoda **`accept`** care specifica criteriul de filtrare si anume, testeaza daca numele fisierului primit ca parametru îndeplineste conditiile dorite de noi.

Definitia interfetei:

```
public interface FilenameFilter       {
        // Metode
        public boolean accept( File dir, String numeFisier );
}
```

Asadar orice clasa de specificare a unui filtru care implementeza interfata `FilenameFilter` trebuie sa implementeze metoda `accept` a acestei interfete. Aceste clase mai pot avea si alte metode, de exemplu un constructor care sa primeasca criteriul de filtrare, adica masca dupa care se filtreaza fisierele. In general, o clasa de specificare a unui filtru are urmatorul format:

```
class DirFilter implements FilenameFilter {
        String filtru;

        //constructorul
        DirFilter(String filtru) {
                this.filtru = filtru;
        }

        //implementarea metodei accept
        public boolean accept(File dir, String nume) {
```

```
                //elimin informatiile despre calea fisierului
                String f = new File(nume).getName();
                if (filtrul este indeplinit)
                        return true;
                else
                        return false;
  }
}
```

Metodele cele mai uzuale ale clasei `String` folosite pentru filtrarea fisierelor sunt:

```
boolean endsWith(String s)
//testeaza daca un sir se termina cu sirul specificat s

int indexOf(String s)
//testeaza daca un sirul are ca subsir sirul specificat s
//returneaza 0=nu este subsir, >0=pozitia subsirului
```

Instantele claselor pentru filtrare sunt primite ca argumente de metode de listare a continutului unui director. O astfel de metoda este metoda `list` a clsei `File`:

```
String[] list (FilenameFilter filtru )
```

Observati ca aici interfata este folosita ca un tip de date, ea fiind substituita cu orice clasa care o implementeaza. Acesta este un exemplu tipic de transmitere a unei functii (functia de filtrare accept) ca argument al unei metode.

Listarea fisierelor din directorul curent care au extensia .java

```
import java.io.*;
public class DirList2 {
  public static void main(String[] args) {
        try {
                File director = new File(".");
                String[] list;
                list = director.list(new FiltruExtensie("java"));

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

Exemplu de folosire a claselor anonime

In cazul în care nu avem nevoie de filtrarea fisierelor dintr-un director decât o singura data, pentru a evita crearea unei noi clase care sa fie folosita pentru filtrare putem apela la o clasa interna anonima, aceasta situatie fiind un exemplu tipic de folosire a acestora.

```
import java.io.*;
public class DirList3 {
  public static void main(String[] args) {
          try {
                  File director = new File(".");
                  String[] list;

                  //folosim o clasa anonima pentru specificarea filtrului
                  list = director.list(new
                          FilenameFilter() {
                                  public boolean accept (File dir,String nume)
                                          return ( nume.endsWith(".java"));
                                  }
                          } );

                  for(int i = 0; i < list.length; i++)
                          System.out.println(list[i]);
          } catch(Exception e) {
                  e.printStackTrace();
          }
  }
}
```