INTERFEŢE

Noţiunea de interfaţă

Aşa cum s-a menţionat anterior, este posibil să declarăm clase abstracte în care toate metodele sunt abstracte. Acest lucru poate fi realizat şi prin intermediul *interfeţelor*. În plus, ele reprezintă mecanismul propus de Java pentru tratarea problemelor ridicate de moştenirea multiplă.

Interfeţele reprezintă o modalitate de a declara un tip constând numai din constante şi din metode abstracte (fără corp). Din versiunea 8 interfețele pot conține și metode cu corp, care sunt fie statice fie au obligatoriu modificatorul **default** (metode default). Ca sintaxă, o interfaţă este asemănătoare unei clase, cu deosebire că în loc de **`class`** trebuie precizat **`interface`**, iar metodele abstracte nu au corp, acesta fiind înlocuit cu `';'`.

Putem spune că interfeţele reprezintă numai **proiecte**, pe când clasele abstracte sunt o combinaţie de **proiecte şi implementări**.

Ca şi în cazul claselor abstracte, este evident că **nu pot fi create obiecte de tipul unei interfeţe.**

Câmpurile unei interfeţe au în mod implicit modificatorii **`static`** şi **`final`**, deci sunt constante.

Metodele abstracte şi default (implicite) din interfeţe sunt totdeauna publice. Metodele abstracte au în mod implicit modificatorul **`abstract`**, iar pentru metodele default trebuie specificat explicit modificatorul **`default`**.

Orice interfaţă este gândită pentru a fi ulterior *implementată* de o clasă, în care metodele interfeţei să fie redefinite, adică să fie specificate acţiunile ce trebuie întreprinse. Faptul că o clasă `C` implementează o interfaţă `I` trebuie specificat prin inserarea informaţiei **`implements`** `I` în antetul clasei.

Exemple de utilizare a interfeţelor

1. Proiecte de clase

```
interface ISort{
    int DIM_MAX=100;//constanta static final - implicit
    void sort();//public abstract
    void afis();
}

class SortJava implements ISort{
    private float f[];
    private int n;
    SortJava(float f[], int n){
```

```
            if(n<DIM_MAX){this.f=f; this.n=n;}
            else{ this.f=null; this.n=0;}//Nu DIM_MAX=n
      }
      public void sort(){  //obligatoriu public
             Arrays.sort(f);
     }
      public void afis(){
            for(int i=0;i<n;i++)
                  System.out.printf("%.2f ",f[i]);
            System.out.println();
      }
  }
```

Utilizare

```
  float f[]={1,5,2,4};
  int nf=f.length;

  ISort i1=new SortJava(f,nf);//declar de tip interfata,aloc clasa
  i1.sort();
  i1.afis();
  //i1.sum(); -nu se poate

  SortJava s=(SortJava)i1;
  System.out.println(s.sum());
```

2. Interfeţele permit transmiterea metodelor ca parametri

```
interface Functie{
      double f(double x);
}
class F1 implements Functie{
      double a,b,c;
      F1(int a1,int b1, int c1){ a=a1;b=b1;c=c1; }
      public double f(double x){ return a*x*x+b*x+c; }
}
class F2 implements Functie{
      public double f(double x){return Math.sin(x); }
}
class MainF{
      static double f(Functie ob, double x){
            return ob.f(x);
      }
      public static void main(String arg[]){
            System.out.println(f(new F1(1,1,1),2));
            System.out.printf("%.2f",f(new F2(),Math.PI/2));
```

```
            //implementarea interfetei Functie cu clasa anonima
            System.out.println(f(new Functie(){
                  public double f(double x){
                        return x*x;}},2));

            //lambda-expresie - versiunea 8
            System.out.println(f((x)->{return x*x*x;},2));
      }
}
```

Precizări:

- dacă o clasă implementează doar unele din metodele abstracte ale unei interfeţe, atunci ea trebuie declarată cu **abstract**;

- o clasă poate implementa oricâte interfeţe, dar poate extinde o singură clasă.

- o interfaţă poate extinde oricâte interfeţe. **Facilitatea ca interfeţele şi clasele să poată extinde oricâte interfeţe reprezintă modalitatea prin care Java rezolvă problema moştenirii multiple.**

- dacă o clasă extinde o altă clasă şi implementează una sau mai multe interfeţe, atunci trebuie anunţată întâi extinderea şi apoi implementarea, ca în exemplul următor:

```
class C extends B implements I1,I2,I3;
```

Exemplu (opțional)

```
interface A {
  int x=1;
  void scrie();
}
interface B extends A {
  int x=2;
  void scrieB();
}
class C {
  int x=3;
  public void scrie() { System.out.println(A.x+" "+B.x+" "+x); }
}
class D extends C implements B {
  int x=4;
  void printx(){ System.out.println(x+" "+super.x); }
  public void scrieB(){System.out.println("B"); }
}
class Interfete2 {
  public static void main(String[] w) {
```

```
    D ob = new D();
    ob.scrie();
    System.out.println(ob.x);
    ob.printx();
    ob.scrieB();
    A ob1 = new D();
    ob1.scrie();
  }
}
```

produce la ieşire:

```
1 2 3
4
4 3
B
1 2 3
```

Interfeţe din Java

1. Interfaţa Comparable - „ordinea naturală" a obiectelor

Interfaţa `Comparable<T>` este din pachetul java.lang şi are o singură metodă

```
int compareTo(T o)
```

Exemplu

```
import java.util.*;
class Persoana implements Comparable<Persoana>{
      String nume;
      int varsta;
      public int compareTo(Persoana persoana){
            return varsta-persoana.varsta;
      }
      //este recomandat sa fie rescrise si equals si hashCode
}
//utilitate
Persoana vp[]={p1,p2, new Persoana("c",15)};
for(Persoana p:vp)
    System.out.println(p);
Arrays.sort(vp);
System.out.println("Dupa sortare ");
for(Persoana p:vp)
```

```
System.out.println(p);
```

Recomandare: Ordinea naturală pentru o clasă `C` trebuie să fie ***consistentă faţă de metoda equals*** adică `e1.compareTo(e2) == 0` ⇔ `e1.equals(e2)` pentru orice două obiecte `e1` şi `e2` de tip `C`.

2. Interfaţa Iterator

- **iterator** = un obiect care permite parcurgerea tuturor elementelor unei colecţii, unul câte unul, indiferent de implementare (având ca tip o clasă care implementează interfaţa `Iterator`)
- pentru a accesa elementele cu ajutorul unui iterator, clasa trebuie să aibă o metodă care furnizează iteratorul, mai exact să implementeze interfaţa `Iterable`
- Interfaţa `Iterable<T>` din pachetul `java.util` are o metodă abstractă

  ```
  Iterator<T> iterator()
  ```

  (plus metode default, printre care **`forEach`**)

- Interfaţa `Iterator<E>` din pachetul `java.util` are metodele abstracte

  ```
  boolean hasNext()
  E next() //throws NoSuchElementException
  ```

  (plus metode default)

Exemplu În exemplul de mai jos este implementată o colecţie de caractere iterabilă, memorată intern ca **`String`** (exemplul fiind didactic, pentru a ilustra ideea de iterator). Pentru a simplifica accesul clasei **`IteratorSir`** la membrii clasei **`Sir`** clasa **`IteratorSir`** este clasă internă a clasei **`Sir`**.

```
import java.util.*;
class Sir implements Iterable<Character> {
      private String s;
      Sir(){}
      Sir(String s1){ s=s1 ;}
      Sir(char[] s1){ s=new String(s1);}
      public Iterator<Character> iterator(){
            return new IteratorSir();
      }

      class IteratorSir implements Iterator<Character>{
            private int curent=0;
            public boolean hasNext(){
                  return curent<s.length();
            }
            public Character next(){
                  if (curent==s.length())
```

```
                        throw new NoSuchElementException();
                  char c= s.charAt(curent);
                  curent++;
                  return c;
            }
            public void remove() {
                throw new UnsupportedOperationException();
            }
      }
}
class ExpIterat{
      public static void main(String aa[]){
            char vc[]={'x','y','z'};
            Sir sc=new Sir(vc);
            for(Character c:sc)
                  System.out.println(c);
            Sir s=new Sir("abcdef");
            for(Character c:s)
                  System.out.println(c);
            for(char c:s)//merge si cu char, unboxing
                  System.out.print(c);
            System.out.println();
            Iterator<Character> it=s.iterator();
            while(it.hasNext())
                  System.out.println(it.next());
            //System.out.println(it.next());    //NoSuchElementException
      }
}
```

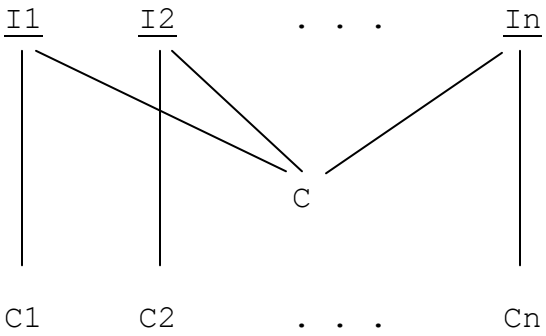
Observaţie. Puteam scrie clasa **`IteratorSir`** ca o clasă exterioară clasei **`Sir`**, dar pentru ca aceasta să poată accesa membrii clasei **`Sir`** trebuie să adăugăm în clasa **`IteratorSir`** un câmp de tip **`Sir`** şi un constructor care primeşte ca parametru un **`Sir`**, iar metoda `iterator`() va fi:

```
      public Iterator<Character> iterator(){
            return new IteratorSir(this);
      }
```

- ## Rezolvarea în Java a problemei moştenirii multiple (opțional)

Să presupunem că plecând de la clasele `C1, C2, ..., Cn` dorim să construim o nouă clasă care să moştenească unele dintre metodele lor. Java permite doar moştenire simplă, deci va fi necesar să apelăm la interfeţe.

Modalitatea de rezolvare a problemei moştenirii multiple pe care o prezentăm în continuare este independentă de numărul de clase moştenite. Vom folosi următoarea structură de interfeţe şi clase:



În această structură clasele `C1, C2, ..., Cn` implementează respectiv metodele anunţate în interfeţele `I1, I2, ..., In`, iar clasa `C` implementează toate interfeţele `I1, I2, ..., In`. Este suficient să prezentăm modul în care clasa `C` moşteneşte implementările din `C1` ale metodelor anunţate în `I1`, lucrurile decurgând analog pentru celelalte interfeţe şi clase pe care le implementează. De aceea *în continuare vom presupune că* `C` *trebuie să extindă doar clasa* `C1`.

În clasa `C` vom declara şi crea un obiect `ob1` de tipul `C1` (se presupune că în clasa `C` se ştie ce implementare a interfeţei `I1` trebuie folosită). Atunci pentru fiecare metodă `met` implementată de `C1` introducem în clasa `C` metoda `met` cu aceeaşi signatură şi având una dintre formele:

tip `met(...) { return ob1.met(...); }`

`void met(...) { ob1.met(...); }`

după cum metoda întoarce sau nu o valoare.

Exemplul 1

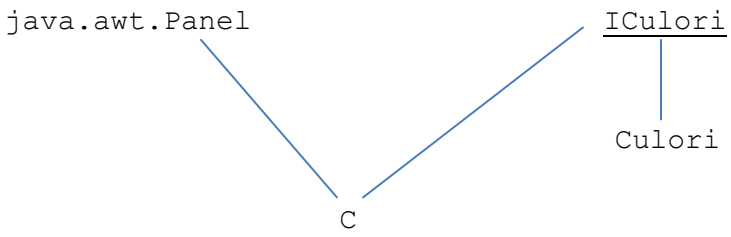
```
interface X {
  void x1();
  int x2();
}

class CX implements X {
  public void x1() { System.out.print("x1 "); }
  public int x2() { return 1; }
}

class C implements X {
  X obX= new CX(); // Clasa C "stie" ca trebuie sa foloseasca
```

```
                    // implementarea CX a interfetei X
  public void x1() { obX.x1(); }
  public int x2() { return obX.x2(); }
}
```

Exemplul 2




```
import java.awt.*;
interface ICulori {
  void next();
  int get();
}
class Culori implements ICulori {
  int cul;
  public void next() { cul=cul+1; }
  public int get() { return 1; }
}
class C extends Panel implements ICulori {
  ICulori obI= new Culori(); //implementare pentru interfata
  public void next() { obI.next(); }
  public int get() { return obI.get(); }
}
class Mult1 {
  public static void main (String[] s) {
    C obC = new C();
    obC.next(); System.out.println(" "+obC.get());
  }
}
```