Lab5: Interfețe

Cuprins

Recapitulare laborator 4

În laboratorul 4, am văzut cum se poate reutiliza codul într-o aplicaţie Java: fie prin agregare, fie prin extinderea claselor, în funcţie de datele problemei analizate. Diferenţa principală dintre cele două metode este dată de cele două tipuri majore de relaţii ce pot exista între obiectele aplicaţiei: “is a” (“*Dog* is an *Animal*)” sau “has a” (“*Car* has an *Engine*).

Moştenirea reprezintă, aşa cum am văzut, posibilitatea de a extinde (cel mult) o clasă deja existentă, preluând sau remodelând funcţionalitatea clasei părinte, dar şi adăugând un nou comportament. De asemenea, am introdus clasele abstracte, ce definesc un model (un concept abstract) ce va fi folosit pentru definirea altor clase ce pot fi instanţiate. O clasă abstractă nu poate fi instanţiată, doar oferă un comportament comun subclaselor sale.

Încercaţi să răspundeţi la următoarele întrebări:

- Ce este agregarea?
- Ce este moştenirea?
- Când se foloseşte agregarea şi când moştenirea?
- Ce este upcastingul şi când se foloseşte?
- Ce este downcastingul şi când se foloseşte?
- La ce este utilă o clasă abstractă?
- La ce trebuie să fim atenţi când extindem o clasă?

Noţiunea de interfaţă

O interfaţă este o colecţie de declaraţii de constante (câmpuri cu modificatorii **`static`** şi **`final`**) şi metode abstracte (fără implementare), stabilind o “formă” (schelet) pentru clasele care o implementează. Intuitiv, interfeţele determină un “contract”, un protocol între clase, respectiv o clasă care implementează o interfaţă trebuie să implementeze metodele definite în acea interfaţă.

O interfaţă este diferită de o clasă prin mai multe lucruri, respectiv:

- o interfaţă nu se poate instanţia;
- o interfaţă nu conţine constructori;
- toate metodele unei interfeţe sunt abstracte;
- câmpurile unei interfeţe sunt implicit constante;
- o interfaţă poate extinde mai multe interfeţe, spre deosebire de clase care nu pot extinde decât o clasă;
- o clasă poate implementa una sau multe interfeţe.

Definirea interfeţelor

Pentru a crea o interfaţă se foloseşte cuvântul cheie **`interface`** în loc de **`class`**:

```
[public] interface NumeInterfaţă [extends SuperInterfaţă1,
SuperInterfaţă2...] {
      // Corpul interfeţei, respectiv:
      // constante şi metode fără implementare
}
```

- La fel ca în cazul claselor, o interfaţă poate fi declarată `public` doar dacă este definită într-un fişier cu acelaşi nume ca şi aceasta;
- Dacă nu se declară cu modificatorul `public`, atunci specificatorul de acces va fi implicit `package-private`.

Observaţii:

- Ca şi în cazul claselor abstracte, nu pot fi create obiecte de tipul unei interfeţe;
- Orice interfaţă trebuie gândită pentru a fi ulterior *implementată* de o clasă, în care metodele să fie definite, adică să fie specificate acţiunile ce trebuie întreprinse;
- Toate metodele unei interfeţe sunt implicit publice şi au modificatorul `abstract`; ele nu pot fi statice, deoarece, fiind abstracte, nu pot fi specifice claselor.
- Implicit, specificatorul de acces pentru membrii unei interfeţe este **public**. Atunci când implementăm o interfaţă trebuie să specificăm că funcţiile sunt `public` chiar dacă în interfaţă ele nu au fost specificate explicit astfel. Acest lucru este necesar deoarece specificatorul de acces implicit în clase este `package-private`, care este **mai restrictiv** decat `public`.

Exemplu:

```
public interface Instrument {

   int CONSTANT_INT = 1; // Constanta; static & final

   void play(); // Automat public
   String what();
   void adjust();

   protected int doSomething(); // Modificator nepermis; Eroare
   private String doSomething2(); // Modificator nepermis; Eroare
}
```

Implementarea interfeţelor

Faptul că o clasă C implementează o interfață I trebuie specificat prin folosirea cuvântului cheie **`implements`** `<NumeInterfață>` în antetul clasei. Astfel clasa este conformă cu interfaţa, "moştenind" scheletul pus la descris de aceasta.

De asemenea, o clasă trebuie să implementeze **toate** metodele specificate de interfață, cu excepţia claselor abstracte. Cu alte cuvinte, o clasă care implementează o interfață se poate declara abstractă şi atunci nu mai este necesar să implementeze toate metodele declarate în interfață.

O clasă care implementează o interfață poate fi folosită în locurile unde se aşteaptă o variabilă de tipul interfeţei. Presupunând că avem o clasă `Chitară` care implementează interfaţa instrument, putem declara:

```
Instrument chitară = new Chitară();
```

Creaţi o interfață `ISort` care să conţină:

- o constantă DIM_MAX = 100;
- două metode abstracte `sort()` şi `afişare()`.

Creaţi, apoi, o clasă `SortJava` care are două câmpuri, unul de tip float[] şi celălalt de tip întreg, reprezentând numărul de elemente ale vectorului. Scrieţi un constructor pentru această clasă, implementaţi cele două metode ale interfeţei pentru a sorta, respectiv afişa vectorul dat ca membru al clasei şi adăugaţi încă o metodă pentru a calcula suma elementelor acestuia.

Definiţi o nouă clasă, `BubbleSort` care implementează, de asemenea, interfaţa `ISort`. Ea va fi similară celelaltei clase, cu exceptia faptului că va conţine un câmp de tip `int[]`, în loc de `float[]`. Verificaţi codul cu ajutorul unei clase ce conţine numai metoda `main`.

Observaţie: Presupunem că avem următoarea clasă ce conţine numai metoda principală (referitoare la exerciţiul de mai sus). Observăm următoarele lucruri:

- *(1)* putem avea variabile declarate de tipul unei interfeţe, dar la iniţializarea lor trebuie să folosim tipul unei clase care implementează interfaţa respectivă
- *(2)* dacă declarăm o variabilă de tipul unei interfeţe, atunci nu putem apela decât metodele definite în acea interfață, chiar dacă în clasa ce implementează interfaţa respectivă sunt definite şi alte metode;
- *(3)* pentru a apela metoda `sum` este necesară conversia explicită la tipul clasei care conţine metoda respectivă şi implementează interfaţa, `SortJava`

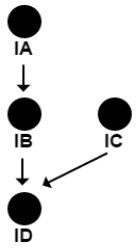
```
public class SortMain {
   public static void main(String args[]) {
        int a[] = {5, 3, 1, 2}, na = a.length;
        float f[] = {1, 5, 2, 4};
        int nf = f.length;
        ISort i1 = new SortJava(f, nf); // (1) tip interfață
        i1.sort();
        i1.afis();
        //i1.sum(); // (2) - nu se poate

        SortJava s = (SortJava) i1; // (3)conversie de tip
        System.out.println(s.sum());
             i1 = new BubbleSort(a, na); // alta implementare a
interfetei
        i1.sort();
        i1.afis();
   }
}
```

Extinderea interfețelor

Ca şi clasele, interfețele pot fi extinse. O interfață `I` poate extinde oricâte interfețe, în acest mod adăugându-se la I noi constante şi (anunțuri de) metode. În Java, este permis ca o interfață ce extinde o alta interfață să conțină o constantă cu același nume.

Presupunem că avem următoarea structură de interfețe şi că o constantă `c` este declarată într-una sau mai multe interfețe.



Putem deosebi două cazuri:

1) Constanta `c` este redeclarată în interfața `ID`: o referire la `c` constituie o referire la constanta `c` din `ID`. Putem face însă referire şi la constantele din celelalte interfețe prin `IA.c`, `IB.c` şi `IC.c`.

2) Dacă `c` este declarată în mai multe interfețe, atunci când este folosită constanta `c` într-o clasă ce implementează interfața `ID`, aceasta se va referi la cea mai recentă declarare a ei. Dacă `c` este declarată în `IB` şi `IC` şi nu este redeclarată în `ID`, atunci la folosire, trebuie specificată explicit din ce interfață să se citească `c`. Acest lucru se întâmplă deoarece `IB` şi `IC` se află pe același nivel în ierarhie.

În cazul metodelor, dacă în interfețele `IB` și `IC` (sau în supertipuri ale lor) apare o metodă cu același nume, atunci avem situațiile:

1) dacă metodele au signaturi diferite, vor fi moștenite ambele metode;
2) dacă metodele au aceeași signatură și același tip pentru valoarea întoarsă, va fi moștenită o singură metodă;
3) dacă metodele au aceeași signatură, dar tipurile valorilor întoarse diferă, atunci moștenirea nu va fi posibilă (eroare de compilare).

Exemplu:

```
interface IA {
   char c = 'a';
}

interface IB extends IA {
   int c = 1;
   void met();
}

interface IC {
   boolean c = true;
   void otherMet();
}

interface ID extends IB, IC {
   int c = 99;
   void met();
}

class B implements IB {
   public void met() {
       System.out.println("++++++");
   }
}

class D implements ID {
   public void met() {
       System.out.println("******");
   }

   public void otherMet() {
       System.out.println("------");
   }
}

class MainInterf {
```

```
    public void met() {
    }

    public static void main(String[] s) {
        B ob1 = new B();
        ob1.met();
        D ob2 = new D();
        ob2.met();
        ob2.otherMet();
        int c = -88;
         System.out.println(c + " " + IA.c + " " + IB.c + " " + IC.c
+ " " + ID.c);
    }
}
```

Codul de mai sus va produce la ieşire:

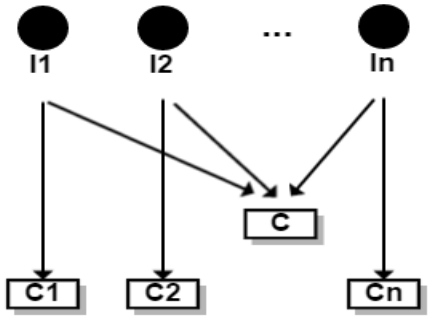
```
++++++
******
------
abc
-88 a 1 true 99 1 99
```

Problema moştenirii multiple în Java

Moştenirea multiplă este facilitatea oferită de unele limbaje de programare ca o clasă să moştenească (prin extindere) membri ai mai multor clase. Evident, această caracteristică importantă a programării orientate pe obiecte nu este neglijată în Java.

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 interfețe.

Problema moştenirii multiple se poate rezolva în Java prin următoarea structură de interfețe şi clase:



Clasele `C1, C2, ..., Cn` implementează respectiv metodele interfețelor `I1, I2, ..., In`, iar clasa `C` implementează toate interfețele `I1, I2, ..., In`.

Exemplu: Rularea codului următor va afișa:

```
CX1...
 2
```

```
public interface X {
    void met1();
    int met2();
}

class CX1 implements X {
    @Override
    public void met1() {
        System.out.println("Method1...");
    }

    @Override
    public int met2() {
        return 1;
    }
}

class CX2 implements X {
    @Override
    public void met1() {
        System.out.println("CX2...");
    }

    @Override
    public int met2() {
        return 2;
    }
}

class C implements X {
    X ob1, ob2;

    C(X ob1, X ob2) {
        this.ob1 = ob1;
        this.ob2 = ob2;
    }

    @Override
    public void met1() {
```

```
        ob1.met1();
    }

    @Override
    public int met2() {
        return ob2.met2();
    }
}

public class Main {
    public static void main(String[] args) {
        X ob1 = new CX1();
        X ob2 = new CX2();
        C obC = new C(ob1, ob2); // (1)
        obC.met1();
        System.out.print(" " + obC.met2());
    }
}
```

Observaţie:

- Amintim că putem declara variabile având ca tip numele unei interfețe şi putem atribui unei astfel de variabile un obiect care implementează interfața.
- În exemplul de mai sus nu este necesar ca `C` să ştie care este clasa ce implementează pe `I1`, ci poate afla acest lucru prin intermediul unui constructor; cu alte cuvinte, la crearea unei instanțieri a lui `C`, putem preciza ce implementare a lui `I1` să folosească.

Interfeţe vs. clase abstracte

Clasele abstracte şi interfeţele nu se exclud reciproc. Utilizarea unei clase abstracte sau a unei interfeţe depinde de problema curentă, însă următoarele lucruri trebuie, totuşi, luate în considerare:

- dacă o clasă extinde o clasă abstractă, atunci ea nu va mai putea extinde o altă clasă; în acest caz, mai ales în situaţii în care clasa curentă este într-o relaţie de tipul “is-a” cu o altă clasă (deci este necesară moştenirea), se recomandă folosirea interfeţelor;
- dacă o clasă (neabstractă) implementează o interfaţă, atunci ea trebuie să implementeze toate metodele definite în interfaţă; în cazul în care numărul acestor metode este unul mare, iar programatorul nu are nevoie decât de foarte puţine dintre acestea, atunci se poate folosi o clasă abstractă care să conţină numai metodele necesare;

- evident, în unele situaţii este necesar ca o clasă să implementeze mai multe interfeţe, dar să şi extindă o clasă.

Dăm, în continuare, unele dintre cele mai importante diferenţe dintre o clasă abstractă şi o interfaţă.

Clasă abstractă	**Interfaţă**
nu se impune ca toate câmpurile să fie constante	toate câmpurile sunt constante (static final)
pot exista membri protected sau private	toţi membrii sunt implicit public
nu toate metodele este obligatoriu să fie abstracte	toate metodele definite sunt abstracte
pot extinde o singură clasă abstractă	pot extinde oricâte interfeţe

Alte exemple de folosire a interfeţelor

Interfeţele permit, de asemenea, *trimiterea metodelor ca parametrii.* Urmăriţi exemplul următor:

Exemplu:

```
public 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 cheamaF(Functie ob, double x) {
        return ob.f(x);
    }

    public static void main(String arg[]) {
        System.out.println(cheamaF(new F1(1, 1, 1), 2));
        System.out.printf("%.2f", cheamaF(new F2(), Math.PI / 2));
    }
}
```

Observaţie:
Java permite crearea unei clase anonime care să implementeze interfaţa. De exemplu, folosind interfaţa `Funcţie` declarată mai sus, putem scrie:

```
public static void main(String arg[]) {
    Functie cos = new Functie() {
        @Override
        public double f(double x) {
            return Math.cos(x);
        }
    };
    System.out.println(cos.f(Math.PI / 2));
}
```

Aşa cum ştim, interfeţele nu pot fi instanţiate, însă în acest caz Java creează o clasă anonimă care implementează acea interfaţă.

Acest concept este folosit, aşa cum vom vedea, pentru a "lega" funcţii de anumite evenimente (callbacks). Acesta este des folosit în dezvoltarea interfeţei grafice.

Exemple de interfeţe din Java

Interfaţa Comparable

Interfaţa `Comparable` se găseşte în pachetul `java.lang` şi are o singură metodă:

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

Aceasta poate fi folosită pentru a sorta obiectele după anumite atribute, impunând o relaţie de ordine totală asupra obiectelor unei clase ce implementează interfaţa. De exemplu, putem sorta persoanele după vârstă sau angajaţii unei companii după salariu.

Rezultatul întors de metoda `compareTo` poate fi:

- un întreg negativ, dacă obiectul curent este "mai mic" decât obiectul dat ca paramtru;
- zero, dacă obiectul curent este "egal cu" obiectul dat ca paramtru;
- un întreg pozitiv, dacă obiectul curent este "mai mare" decât obiectul dat ca parametru.

 **Observație:**

Ordinea naturală pentru o clasă `C` trebuie să fie consistentă față de metoda `equals` adică `e1.compareTo(e2) == 0` `e1.equals(e2)` pentru orice două obiecte `e1` și `e2` de tip `C`.

 **Exemplu:**

```
public class Persoana implements Comparable<Persoana>{
    String nume;
    int varsta;

    public Persoana(String nume, int varsta) {
        this.nume = nume;
        this.varsta = varsta;
    }

    @Override
    public int compareTo(Persoana p) {
        return this.varsta - p.varsta;
    }

    @Override
    public boolean equals(Object o) { // compatibil cu compareTo
        if(o == null) {
            return false;
        }
        if(o instanceof Persoana) {
            Persoana persoana = (Persoana) o;
            return varsta == persoana.varsta;
        }
        return false;
    }

    @Override
    public int hashCode() {
        int result = 17;
        result = 31 * result + varsta;
        return result;
```

```
    }
}
```

Observaţie: Pentru compararea obiectelor după mai multe criterii, se foloseşte interfaţa `Comparator`.

Exerciţii

1. Implementaţi interfaţa `Task` (dată mai jos) în cele 3 moduri.

```
public interface Task {
    // Execută acțiunea specifică taskului
    void execute();
}
```

- Un task care afişează un mesaj la output. Mesajul este dat în constructor.
- Un task care reţine data la care a fost creat taskul şi se afişează un mesaj cu această oră. Timpul se consideră cel din momentul în care este apelat un constructor.
- Un task care contorizează numărul de instanţe generate pentru acel task. Contorul va fi afişat după fiecare incrementare.

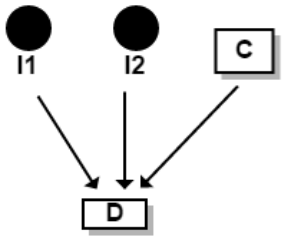
2. Scrieţi o interfaţă `Food` cu metodele:
   - `getCalories()`
   - `getName()`

   Scrieţi o interfaţă `Animal` care să cuprinda metodele:
   - `getName()`
   - `eat(Food food)`

   Implementanţi aceste interfeţe în cadrul claselor:
   - `Cat` şi `CatFood`
   - `Dog` şi `DogFood`

   Nu permiteţi unei pisici să mănânce mâncare de câini (dar invers e posibil). Observaţi că atât `Food` cât şi `Animal` au metoda `getName`, extrageţi-o într-o altă interfaţă `Named` şi extindeţi din ea.

3. Creaţi o structură de clase şi interfeţe după următoarea diagramă, unde `I1` şi `I2` sunt interfeţe, `C` este o clasă, iar `D` implementează interfeţele `I1` şi `I2` şi extinde clasa `C`.



Ce se întâmplă dacă în interfaţa `I1` sau `I2` şi în clasa `C` există un câmp (constantă în cazul interfeţelor) cu acelaşi nume ca în clasa `C`? Dar o metodă? Experimentaţi!

Observaţie (coliziuni de nume):

Revedeţi extinderea interfeţelor din acest laborator. Dacă o metodă **cu acelaşi nume** şi **aceeaşi signatură** apare în superclasă şi în interfețele implementate, atunci:

- dacă precizăm în `C` o implementare a metodei, ea va constitui o redefinire a metodei din superclasă; la aceasta din urmă putem face însă apel prin super;

- dacă în `C` nu este precizată o implementare a metodei, atunci metoda (moştenită) din superclasă constituie implementarea metodei din interfețe, cu condiția ca ea să aibă modificatorul `public`.

4. Ilustrați utilitatea interfețelor în Java şi modul în care se rezolvă problema moştenirii multiple cu ajutorul interfețelor prin nişte exemple concepute de voi.

5. Scrieţi o nouă clasă care să implementeze interfaţa `Funcţie` de mai sus şi să întoarcă logaritmul natural al numărului dat ca parametru.