Interfețe

1. Scopul lucrării

Obiectivele de învățare ale acestei lucrări sunt

- Înțelegerea conceptului de interfață și a diferențelor față de moștenire
- Acumularea de experiență de programare în implementarea interfețelor existente și în definirea și implementarea celor definite de către programator

2. Ce este o interfață

2.1. Definiție și caracteristici

O interfață este o listă de metode care trebuie definite de către orice clasă care implementează interfața respectivă. O interfață poate defini şi constante (public static final).

Seamănă cu o clasă abstractă. O interfață este asemănătoare cu o clasă abstractă care nu are variabile instanță şi statice (constantele static final sunt permise) şi fără corpuri pentru metode. Aceasta este în esență ceea ce face o clasă complet abstractă, dar clasele abstracte permit definiții de metode statice, pe când interfețele nu permit acest lucru.

Obligație contractuală. Atunci când o clasă specifică faptul că ea implementează o interfață, clasa trebuie să definească toate metodele respectivei interfețe. O clasă poate implementa mai multe interfețe diferite. Daca o clasă nu defineşte toate metodele interfețelor pe care a fost de acord să le definească (prin clauza *implements*), compilatorul dă un mesaj de eroare, care tipic spune aproximativ "*This class must be declared abstract*" (O clasă abstractă este una care nu implementează toate metodele pe care le-a declarat). Soluția pentru eroare este aproape întotdeauna să se implementeze metodele lipsă din interfață. Un nume de metodă scris greşit sau o listă incorectă de parametri este cauza uzuală, nu faptul că respectiva clasă ar fi trebuit să fie abstractă!

O utilizare foarte frecventă a interfețelor este pentru ascultători (*listeners*). Un ascultător (*listener*) este un obiect dintr-o clasă care implementează metodele cerute pentru interfața respectivă. Puteți crea ascultători interiori (*inner*) anonimi sau puteți implementa interfața cerută în oricare clasă. Interfețele sunt de asemenea folosite extensiv în pachetul de structuri de date (*Java Collections*).

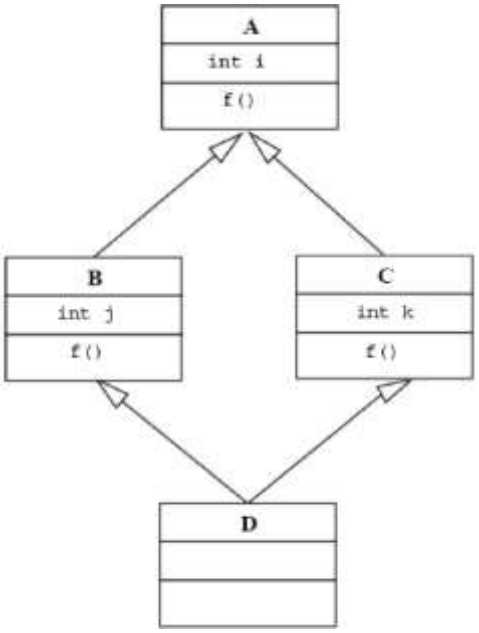
2.2. Clasele în comparație cu interfețele

Clasele sunt folosite pentru a reprezenta ceva care are **atribute** (variabile, câmpuri) şi **capabilități/responsabilități** (metode, funcționalități).

Interfețele descriu doar capabilități. Spre exemplu, suntem oameni pentru că avem atributele unui om (clasă). Cineva este instalator pentru că are abilitatea unui instalator (interfață). Cineva poate fi şi electrician (interfață). Puteți implementa multe interfețe, dar obiectele pot fi de o singură clasă.

Această analogie nu merge, totuşi, într-un anume sens. Capabilitățile (metodele) unei clase nu se schimbă (dacă o clasă implementează o interfață, ea este implementată pentru toate instanțele), pe când îndemânările omeneşti despre care vorbeam sunt dinamice şi pot fi învățate sau uitate. Analogia are un defect, cum sunt toate analogiile de altfel, dar oferă o idee despre cum se poate face o distincție între clase şi interfețe.

Interfețele înlocuiesc moştenirea multiplă. O clasă din C++ poate avea mai mult de o clasă părinte.



Aceasta se numeşte moştenire multiplă. Gestionarea definițiilor variabilelor instanță în cazul moştenirii multiple poate deveni destul de încurcată şi duce la mai multe probleme – d.e., aşa numitul "Diamant ucigător al morții" ("*Deadly Diamond of Death*") – decât soluții. Figura din stânga constituie o ilustrare a problemei menționate. În figură sunt prezentate patru clase aranjate într-o structură care creează nevoia moştenirii virtuale. Atât clasa B cât şi clasa C moştenesc din clasa A. D moştenește multiplu atât din B cât şi din C. De aici apar două probleme. Prima: ce implementare a metodei 'f' să moştenească D? Ar trebui să moştenească f() din B sau f() din C? În C++ răspunsul se dovedeşte a fi nici una. Trebuie declarat f() în D şi trebuie implementat. Aceasta elimină ambiguitatea şi cu siguranță această regulă simplă ar fi putut fi adoptată în Java.
Totuşi, cea de a doua problemă este un pic mai complicată. Clasa A are o variabilă membru numită i. Amândouă clasele B şi C moştenesc aceasta variabilă membru. Cum D o moştenește de la amândouă, avem de-a face cu o ambiguitate. Pe de o parte am dori ca i din B şi i din C să fie variabile separate în D, creând astfel două copii ale lui A în D. Pe de altă parte am dori să existe o singură copie a lui A în D astfel ca numai un i din A să existe în D.
Din acest motiv proiectanții limbajului Java au ales să permită doar o singură clasă părinte, dar multiple interfețe. Aceasta furnizează majoritatea funcționalității moştenirii multiple, dar fără dificultățile pe care le incumbă.

2.3. Definirea și implementarea unei interfețe

2.3.1. Sintaxa:

```
[modificator acces] interface NumeInterfata{
        //constante
        public static final X = 10;
        int Y = 20; // pare o variabila instanta, dar este o constanta
                    // implicit este declarata cu modificatorii public static final
        public void numeMetoda();
}
```

2.3.2. Implementarea unei interfețe:

```
class NumeClasa implements NumeInterfata{
        public void numeMetoda(){
            // codul necesar pentru implementarea metodei
        }
}
```

2.3.3. Instanțierea unei interfețe

La fel ca și în cazul claselor abstracte, o interfață nu poate fi instanțiată. Prin definiție, o interfață nu poate avea constructori dar putem declara o variabila de tipul NumeInterfata și o putem instanția cu orice clasă concretă care implementează interfațat respectivă. De exemplu:

```
NumeInterfata v = new NumeClasa();
```

Acest fel de instanțiere a unei variabile de un anumit tip cu o instanță de alt tip produce *polimorfismul*. Interfețele Java sunt o modalitate de a obține polimorfism. Tipul obiectului în cazul nostru este o interfață, dar în general poate fi atât o clasă cât și o interfață.

2.3.4. Extinderea unei interfețe prin moștenire

Interfețele se pot extinde prin moștenire. Un exemplu în acest sens:

```
//Filename: Sports.java
public interface Sports
{
        public void setHomeTeam(String name);
        public void setVisitingTeam(String name);
}

//Filename: Football.java
public interface Football extends Sports
{
        public void homeTeamScored(int points);
        public void visitingTeamScored(int points);
        public void endOfQuarter(int quarter);
}

//Filename: Hockey.java
public interface Hockey extends Sports
{
        public void homeGoalScored();
        public void visitingGoalScored();
        public void endOfPeriod(int period);
        public void overtimePeriod(int ot);
}
```

2.3.5. O clasă poate implementa oricâte interfețe

Exemplu de implementare a mai multor interfețe predefinite
Se pot implementa oricâte interfețe într-o clasă; cele pe care doriți să le implementați le separați în clauza *implements* prin virgule. Spre exemplu:

```
//Nota:
//ActionListener are nevoie sa fie definita metoda actionPerformed(..)
//MouseMotionListener are nevoie de definirea lui mouseMoved(..) si mouseDragged(..).
public class MyPanel extends JPanel implements ActionListener, MouseMotionListener {
       public void actionPerformed(ActionEvent e) {
              /* Corpul metodei */
       }
       public void mouseDragged(MouseEvent me) {
              /* Corpul metodei */
       }
       public void mouseMoved(MouseEvent me) {
              /* Corpul metodei */
       }
       // Tot ce mai este in aceasta clasa.
}
```

Este obişnuit ca un panou (*panel*) care foloseşte grafică şi răspunde la mouse să-şi implementeze ascultători pentru evenimentele de mouse proprii (dar nu ascultători pentru acțiuni – *action listeners*) ca mai sus.

Exemplu de declarare a unei interfețe predefinite

Pentru programe simple este mai probabil să folosiți o interfață decât să o definiți. Iată cam ce conține definiția interfeței java.awt.ActionListener.

```
public interface ActionListener{
      public void actionPerformed(ActionEvent e);
}
```

2.4. Câteva interfețe Java predefinite

2.4.1. Interfața Cloneable

Interfața Cloneable este un exemplu de interfață Java predefinită:

- Nu conține antete de metodă sau constante
- Se foloseşte pentru a indica în ce mod trebuie folosită şi redefinită metoda *clone* (moştenită din clasa Object)
- Metoda Object.clone() face o copie bit-cu-bit a datelor obiectului
- Dacă toate datele sunt tipuri primitive sau clase care nu se schimbă (cum este String), atunci acest lucru este adecvat
- Dacă datele din obiectul de clonat includ variabile instanță al căror tip este mutabil, atunci simpla implementare a lui clone poate cauza o scurgere de informații private (privacy leak)
- La implementarea interfeței Cloneable pentru o clasă procedați astfel:
  - Invocați mai întâi metoda clone a obiectului din clasa de bază Object (sau ce altă clasă este)
  - Apoi resetați valorile pentru orice alte variabile instanță noi ale căror valori sunt tipuri de clase mutabile. Aceasta se face prin copierea variabilelor instanță apelând metodele clone proprii
  - Observați că aceasta va funcționa corect doar dacă interfața Cloneable este implementată corespunzător pentru clasele de care aparțin variabilele instanță respective şi pentru clasele de care aparțin oricare dintre variabilele instanță din clasele menționate și așa mai departe

2.4.2. Interfața Comparable

- Interfața Comparable este definita în pachetul java.lang, fiind automat disponibilă fiecărui program. Nu are decât următoarea metodă care trebuie implementată:

```
public int compareTo(Object other);
```

- Este responsabilitatea programatorului să urmeze semantica interfeței Comparable atunci când implementează. Metoda compareTo trebuie să returneze
  - Un număr negativ atunci când un obiect "este înainte de" parametrul other
  - Zero atunci când obiectul apelant "este egal cu" parametrul other
  - Un număr pozitiv dacă obiectul apelant "urmează după" parametrul other
- Dacă parametrul other nu este de acelaşi tip cu clasa în curs de definire, atunci trebuie aruncată excepție de tipul ClassCastException
- Aproape orice noțiune rezonabilă de "este înainte de" este acceptabilă
  - In particular, toate relațiile standard mai-mic-decât peste numere şi ordonările lexicografice peste şirurile de caractere sunt potrivite
- Relația "urmează după" este doar inversul lui "este înainte de"
- Pot fi luate în considerare alte ordonări, atâta vreme cât sunt relații de ordine totală. O ordine totală trebuie să satisfacă următoarele condiții:
  - (Antireflexivitate) Pentru nici un obiect o, o nu este înaintea lui o

  - (Trihotomi-antisimetrie) Pentru oricare două obiecte o1 şi o2, una şi numai una dintre următoarele este adevărată: o1 este înainte de o2, o1 urmează după o2, sau o1 este egal cu o2
  - (Tranzitivitate) Dacă o1 este înainte de o2 şi o2 este înainte de o3, atunci o1 este înainte de o3
- Semantica lui "equals" din metoda compareTo ar trebui să coincidă cu cea a metodei equals dacă se poate, dar aceasta nu este absolut necesar
- Observație: atât clasa Double cât şi clasa String implementeză interfața Comparable
  - Interfețele se aplică doar claselor
  - Un tip primitiv (d.e., double) nu poate implementa o interfață

Un exemplu pentru Comparable

```
abstract class Employee implements Comparable{

	private String name;

	public Employee(String name)
	{
		this.name = name;
	}

	public String getName()
	{
		return name;
	}

	public int compareTo(Object other)
	{
		Employee e = (Employee) other;
		return name.compareTo(e.name);
	}
	public abstract double calculatePay();
}
```

2.4.3. Interfața Enumeration

Un obiect care implementează interfața Enumeration generează o serie de elemente, câte unul o dată, adică reprezintă un iterator. Apelurile succesive la metoda nextElement() returnează elementele succesive ale seriei iar metoda hasMoreElements() testează dacă enumerarea respectivă mai conține elemente.
Spre exemplu, pentru a tipări toate elementele unui vector v :

```
for (Enumeration e=v.elements(); e.hasMoreElements() ;){
	System.out.println(e.nextElement());
}
```

3. Mersul lucrării

3.1. Studiați materialul şi exemplele furnizate.

3.2. Creati o interfață **Numeric** care să conțină operații de adunare, scădere, înmulțire. Creați apoi următoarele clase care să implementeze interfața Numeric: **Complex**, **Fractie**. Implementați și o clasă **Matrice** care poate să aibă elemente de oricare din cele două tipuri (numere complexe / fracții). În clasa matrice implementați operațiile de adunare, scădere, înmulțire și înmulțire cu scalar. Testați funcționalitatea claselor într-o metodă **main** într-o clasă separată.

3.3 Rezolvați problema tablei de șah din laboratorul precedent definind **PiesaSah** ca interfață în loc de clasă abstractă. Care dintre cele două variante este mai eficientă în cazul acestei probleme? De ce?

3.4. Creați o clasă **Persoana** care să conțină nume, prenume, vârstă. Într-o metodă **main** creați un tablou de obiecte de tip Persoana. Ordonați vectorul în funcție de nume.
Indicații implementare:

- Folosiți metoda sort(...) din clasa predefinită Arrays
- Implementați interfața Comparable în clasa Persoana pentru a specifica criteriul de sortare
- Modificați conținutul metodei compareTo(Object other) pentru a putea sorta vectorul de persoane în funcție de vârstă.

3.5. Creați trei interfețe, fiecare cu două metode. Moșteniți o interfață nouă din cele trei adăugând o nouă metodă. Creați o clasă concretă care să implementeze noua interfață şi să şi moştenească dintr-o altă clasă. Scrieți apoi patru metode, fiecare dintre ele să primească una dintre cele patru interfețe ca argument. În metoda **main**, creați un obiect din clasa concretă și transmiteți-l fiecăreia dintre cele trei metode.