Lucrare de laborator nr. 9
Clase abstracte. Interfeţe. Polimorfism

1. Scopul lucrării

În această lucrare se studiază clasele abstracte şi interfeţele. Se studiază şi polimorfismul la execuţie (*late binding*).

2. Breviar teroretic

Clase abstracte

O clasă **abstractă** este o clasă din care nu se pot instanţia obiecte. Din ea se pot însă deriva noi clase (poate fi subclasată). Ea este folosită ca şi **şablon** pentru a fi moştenit de alte clase.

Clasa abstractă poate să conţină două feluri de metode: metode care au definiţia completă (metode care au şi implementare) şi metode care nu sunt implementate (nu au corp, au doar antet). Metodele care nu sunt implementate se definesc cu ajutorul cuvântului cheie **abstract**. În general, aceste metode abstracte sunt implementate de către clasele derivate din clasa abstractă.

O clasă abstractă se defineşte cu ajutorul cuvântului cheie **abstract**.

Exemplu

```
abstract class A
{
   ..............
}
A obj=new A( ); //eroare de sintaxă!!!
```

Interfeţe

O **interfaţă** conţine o listă de semnături de metode (metode fără implementare). O interfaţă poate să definească, de asemenea, constante. O clasă ce implementează o anumită interfaţă se obligă să dea implementarea pentru **toate** metodele declarate în interfaţă.

O clasă ce implementează o anumită interfaţă foloseşte cuvântul cheie **implements**.

Noţiunea de interfaţă poate fi văzută ca o generalizare a noţiunii de clasă abstractă, în sensul că toate metodele din interfaţă sunt abstracte (fără implementare).

O interfaţă se defineşte cu ajutorul cuvântului cheie **interface**.

O clasă nu poate să moştenească decât o singură clasă, dar poate implementa mai multe interfeţe.

De ce s-a inventat şi noţiunea de interfaţă? S-a constatat că în timp, semnăturile metodelor sunt mai viabile, în sensul că "au o viaţă mai lungă" decât implementările.

Polimorfism

Noţiunea de **polimorfism în faza de execuţie** este legată in primul rand de noţiunea de moştenire.

În cazul unei metode prezentă în clasa de bază şi redefinită în clasa derivată (acelaşi nume, două forme: o formă în clasa de bază, o formă în clasa derivată) în anumite situaţii, numai în faza de execuţie se poate şti care formă de metodă se va apela. Altfel spus, numai la execuţie se poate şti care cod va fi folosit (va fi "legat") de aplicaţie.

Codul metodei adecvate va fi "legat" la execuţie şi aceasta se cheamă **late-binding** (legare târzie), spre deosebire de legarea codului din faza de compilare **(early-binding)**.

3. Probleme rezolvate

Problema 1

Definim clasa abstractă A ce are două metode: una implementată şi alta care nu este implementată.

- metoda abstractă calcul();
- metoda durataCalcul() ce returnează durata exprimată în milisecunde, a execuţiei metodei calcul();

Din clasa abstractă A, se va deriva clasa B ce conţine implementarea metodei calcul(). Se va dezvolta şi o clasă de test, pentru clasa derivată B.

```
abstract class A
{
 abstract public void calcul(int N);
 public long durataCalcul(int N)
```

```
 {
  long t1=System.currentTimeMillis();
  calcul(N);
  long t2=System.currentTimeMillis();
  return (t2-t1);
 }
}
class B extends A
{
 public void calcul(int N)
 {
  //Calculeaza N*N*N produse
  int i,j,k;
  long rezultat;
  for(i=1;i<=N;i++)
   for(j=1;j<=N;j++)
    for(k=1;k<=N;k++)
        rezultat=i*j*k;
 }
}
class Test
{
 public static void main(String args[])
 {
  final int N=1000;
  B b=new B();
  System.out.println("durata calcul = "+b.durataCalcul(N)+" ms.");
 }
}
```

Problema 2

Definim interfaţa Forma în care avem două antete de metode: arieTotală( ) şi volum( ). Din această interfaţă să se implementeze clasa Cub, ce are ca variabilă de instanţă latura cubului.

```
interface Forma
{
  public double arieTotală( );
  public double volum( );
}
```

```
class Cub implements Forma
{
  private double a;
  public Cub(int a)
  {
    this.a=a;
  }
  public double get( )
  {
    return a;
  }
  public void set(int a)
  {
        this.a=a;
  }
  public double arieTotală( )
  {
    return 6*a*a;
  }
  public double volum( )
  {
    return a*a*a;
   }
}
class TestCub
{
        public static void main(String args[])
        {
                Cub c=new Cub(3);
                System.out.println("volum cub = "+c.volum());
        }
}
```

Problema 3

Să se construiască clasa Cerc, ce are ca variabilă de instanţă privată, un număr întreg r, ce reprezintă raza unui cerc. În această clasă avem ca metode:

- constructorul, ce face iniţializarea razei;
- afisare(), ce afişează raza cercului.

Din clasa Cerc se va deriva clasa CercExtins, în care se vor adăuga ca variabile de instanţă x şi y: coordonatele centrului şi se va defini constructorul clasei şi se va redefini metoda afisare() (pe lângă rază, va afişa şi coordonatele centrului)
Folosind cele două clase anterioare, Cerc şi CercExtins să se dezvolte o aplicaţie în care se vor citi N cercuri (de tipul Cerc sau CercExtins), ce se memorează într-un vector. Citirea unui obiect de tip Cerc sau CercExtins este dată de valoarea 0 sau 1 a unui număr aleator generat. În final se vor afişa pentru fiecare cerc din cele N informaţiile memorate (pentru fiecare cerc se va apela metoda afişare()).
Această aplicaţie ilustrează conceptul de polimorfism la execuţie.

```
import java.util.*;
class Cerc
{
  private int raza;
  public Cerc(int x)
  {
   raza=x;
  }
  public void afisare()
 {
   System.out.println("raza="+raza);
 }

}
class CercExtins extends Cerc
{
  private int x;
  private int y;
  public CercExtins(int r,int x0, int y0 )
  {
   super(r);
   x=x0;
   y=y0;
 }

 public void afisare()
{
 System.out.println("raza="+this.getRaza());
```

```
 System.out.println("x="+x);
 System.out.println("y="+y);
 }
}


class AfisareCercuri
{
  public static void main (String args[])
  {
    Scanner sc=new Scanner(System.in);
    System.out.print("N=");
    int N=sc.nextInt();
    Cerc c[]=new Cerc[N];//vectorul de obiecte Cerc (clasa de baza)
    Random r=new Random();
    int i;
    int raza,x,y;
    for(i=0;i<N;i++){
     int caz=r.nextInt(2);
     if(caz==0){
       //citeste datele pentru un obiect Cerc:
      System.out.print("raza=");
      raza=sc.nextInt();
      c[i]=new Cerc(raza);
      }else{
            //citeste datele pentru un obiect CercExins:
             System.out.print("raza=");
             raza=sc.nextInt();
            System.out.print("x=");
             x=sc.nextInt();
             System.out.print("y=");
             y=sc.nextInt();
             c[i]=new CercExtins(raza,x,y);
          }
        }//for
     //Afisare vector:
    for(i=0;i<N;i++)
     c[i].afisare();
   }
  }
```

Problema 4

Definim interfaţa Forma2D în care avem două antete de metode: arie() şi perimetru(). Din această interfaţă să se implementeze clasa Cerc, ce are ca variabilă de instanţă raza cercului, clasa Patrat, ce are ca variabilă de instanţă latura pătratului şi clasa Dreptunghi, ce are ca variabile de instanţă lungimea şi lăţimea dreptunghiului.

Folosind aceste trei clase: Cerc, Patrat şi Dreptunghi, să se dezvolte o aplicaţie în care se vor citi datele pentru N obiecte de tipul Forma2D (de tipul Cerc sau Patrat sau Dreptunghi), ce se memorează într-un vector. Citirea unui obiect de tip Cerc sau Patrat sau Dreptunghi este dată de valoarea 0 sau 1 sau 2 a unui număr generat aleator. În final se vor afişa pentru fiecare obiect din cele N, clasa din care face parte (se va folosi metoda statică *getClass()* din clasa Object), aria şi perimetrul său.

Această aplicaţie ilustrează conceptul de polimorfism la execuţie, în cazul folosirii interfeţelor.

```
import java.util.*;
interface Forma2D
{
  public double arie( );
  public double perimetru( );
}

class Cerc implements Forma2D
{
        private int raza;
        public Cerc(int raza)
        {
                this.raza=raza;
        }
        public double arie()
        {
                return Math.PI*raza*raza;
        }
        public double perimetru()
        {
                return 2*Math.PI*raza;
        }
}
```

```
class Patrat implements Forma2D
{
 private int l;
public Patrat(int l)
{
 this.l=l;
}
  public double arie()
 {
  return l*l;
 }
 public double perimetru( )
 {
  return 4*l;
 }
}

class Dreptunghi implements Forma2D
{
 private int l, L;
public Dreptunghi(int l, int L)
{
 this.l=l;
 this.L=L;
}
 public double arie( )
 {
  return l*L;
 }
 public double perimetru( )
 {
  return 2*l+2*L;
 }
}

class TestPolimorfismForma2D
{
 public static void main(String args[ ])
 {
    Scanner sc=new Scanner(System.in);
```

```
   System.out.print("N=");
   int N=sc.nextInt();
   Forma2D f[]=new Forma2D[N];//vectorul de obiecte Forma2D
   Random r=new Random();
   int i;
   int raza,x,y;
   for(i=0;i<N;i++){
    int caz=r.nextInt(3);
    if(caz==0){
      //citeste datele pentru un obiect Cerc:
      System.out.print("raza=");
      raza=sc.nextInt();
      f[i]=new Cerc(raza);
      }else if(caz==1){
        //citeste datele pentru un obiect Patrat:
        System.out.print("latura patratului = ");
        x=sc.nextInt();
        f[i]=new Patrat(x);
      }else{//este cazul 2 (dreptunghi)
        //citeste datele pentru un obiect Dreptunghi:
        System.out.print("lungimea dreptunghiului = ");
        x=sc.nextInt();
        System.out.print("latimea dreptunghiului = ");
        y=sc.nextInt();
        f[i]=new Dreptunghi(x,y);
      }
   }//for
   for(i=0;i<f.length;i++)
     System.out.println("Obiectul din "+f[i].getClass()+
      " are aria="+f[i].arie()+" si perimetrul="+f[i].perimetru());
  }
}
```

3. Probleme propuse

Problema 1

Pentru interfaţa Forma, definită în problema 2, să se scrie şi clasa Paralelipiped, ce implementează această interfaţă.

Folosind cele două clase ce implementează interfaţa Forma: Cub şi Paralelipiped, să se dezvolte o aplicaţie în care se vor citi datele

pentru N obiecte de tipul Forma (de tipul Cub sau Paralelipiped), ce se memorează într-un vector. Citirea unui obiect de tip Cub sau Paralelipiped este dată de valoarea 0 sau 1 a unui număr generat aleator. În final se vor afişa pentru fiecare obiect din cele N, clasa din care face parte , aria totală şi volumul său.

Problema 2

Să se construiască clasa Punct ce are ca variabile de instanţă două numere întregi x şi y – coordonatele unui punct în plan, şi ca metode:

- Constructorul ce face iniţializările;
- getX() ce returnează valoarea coordonatei x
- getY() ce returnează valoarea coordonatei y
- afisare() în care se afişează coordonatele punctului

Din clasa Punct se derivează două clase: PunctColor şi Punct3D.
Clasa PunctColor faţă de clasa Punct are în plus o variabilă de instanţă în care este memorată culoarea punctului, un nou constructor în care este iniţializată şi culoarea, metoda getCuloare() ce returnează culoarea, şi redefineşte metoda clasei de bază, afişare(), afişând pe lângă coordonatele x şi y şi culoarea.
Clasa Punct3D, ce reprezintă un punct în spaţiu, faţă de clasa Punct are în plus o variabilă de instanţă z, un nou constructor în care sunt iniţializate toate cele trei coordonate, metoda getZ() ce returnează valoarea coordonatei z, şi redefineşte metoda clasei de bază, afişare(), afişând pe lângă coordonatele x şi y şi coordonata z.
Folosind aceste trei clase se va dezvolta o aplicaţie în care se vor citi de la tastatură N puncte (N- citit anterior), de tipul PunctColor sau Punct3D. Pentru fiecare punct, în momentul citirii utilizatorul aplicaţiei va specifica dacă va introduce un PunctColor sau un Punct3D. Cele N puncte se vor memora într-un vector de obiecte de tipul Punct (clasa de bază). În final se vor afişa pentru fiecare punct din cele N informaţiile memorate (pentru fiecare punct se va apela metoda afişare()).