TEHNICI AVANSATE DE PROGRAMARE

LUCRARE DE LABORATOR 4

Variabile shadow. Suprascrierea metodelor. Ascunderea şi încapsularea datelor. Clase şi metode abstracte în Java

I. SCOPUL LUCRĂRII

Lucrarea de față are rolul de a prezenta şi familiariza studentul cu noțiuni privind obiectele şi clasele în limbajului Java: variabile "shadow", suprascrierea metodelor, ascunderea şi încapsularea datelor, clase abstracte şi interfețe; cu importanța şi situațiile de utilizare a acestora.

La sfârşitul acestei lucrări, studentul va avea posibilitatea să scrie programe Java în care să folosească noțiunile menționate anterior.

II. NOȚIUNI TEORETICE

Observație: Vezi noțiunile prezentate în cursul 2.

1. Variabile "shadow"

Consideram următorul exemplu:

```
public class Televizor
{
      int diagonala;
      public String nume;
      static public int an_fabricatie;
      public boolean merge;

      public Televizor(String num,int an,int diag,boolean stare)
      {
        nume=num;
   an_fabricatie=an;
   diagonala=diag;
        merge=stare;
      }

      static int getAnFabricatie()
      {
        return an_fabricatie;
      }

       public void afisare()
      {
         System.out.println("Nume: "+nume);
         if(merge)
         System.out.println("Televizorul este in stare buna de
                             functionare\n");
         else
           System.out.println("Televizorul nu este in stare buna
```

```
                                    de functionare \n");
        }

        public static void main(String args[ ]) {
           System.out.println("Anul fabricatiei este: "
                                +Televizor.getAnFabricatie());
           Televizor t1=new Televizor("Panasonic", 2002, 51, true);
           t1.afisare();
        }
}

public class TelevizorColor extends Televizor
{
        int diagonala;
              public TelevizorColor(String num, int an, int diag,
                                    boolean stare)
        {
          nume=num;
    an_fabricatie=an;
    diagonala=diag;
          merge=stare;
        }
        ... //cititi observatiile de mai jos si accesati variabila
       //diagonala in cadrul acestei clase
}
```

Observații:

1. Observăm declarația în clasa **TelevizorColor** a unei variabile cu numele **diagonala**, care „umbreşte" variabila cu acelaşi nume declarată în clasa **Televizor**.
2. Dacă accesăm numele **diagonala** sau sinonimul **this.diagonala** în cadrul clasei **TelevizorColor** vom accesa variabila **diagonala** declarată în **TelevizorColor**.
3. Altfel, putem sa distribuim (cast) pe **this** clasei corespunzătoare apoi să accesăm **variabila shadow**: ***((Televizor this) .diagonala.***
4. Această tehnică este utilă atunci când în cadrul unei clase se doreşte să se acceseze o variabilă shadow dintr-o clasă care nu este superclasa acesteia.

2. Suprascrierea metodelor în Java

Considerăm următorul exemplu:

```
public class Televizor
{
        int diagonala=51;
        int an_fabricatie=2001;

        int getAnFabricatie()
        {
          return an_fabricatie;
        }
        int getDiagonala()
        {
          return diagonala;
        }
}
```

```
public class TelevizorColor extends Televizor
{
       int diagonala=52;
       int an_fabricatie=2000;

       int getAnFabricatie()
       {
         return an_fabricatie+1;
       }
       int getDiagonala()
       {
         return diagonala-1;
       }

}
...//realizati afisarea anumitor informatii folosind metodele
//descrise mai sus
```

Observații:

1. Am declarat în clasa **TelevizorColor** metodele **getAnFabricatie()** şi **getDiagonala()** având acelaşi nume, tip şi aceleaşi argumente cu metodele din cadrul superclasei **Televizor.** Când aceste metode sunt invocate pentru un obiect al clasei **TelevizorColor** este apelată metoda din cadrul ei şi nu cea a superclasei **Televizor**.
2. Trebuie reținut faptul că suprascrierea metodelor nu este acelaşi lucru cu supraîncărcarea metodelor (în cazul supraîncărcării metodelor este vorba de mai multe metode cu acelaşi nume dar cu liste de argumente diferite).
3. De asemenea, trebuie ştiut faptul că suprascrierea metodelor nu reprezintă, la nivelul metodelor, acelaşi lucru ca şi variabilele shadow

3. Ascunderea şi încapsularea datelor

Java oferă diverşi modificatori utilizați în controlul scopului metodelor şi variabilelor (câmpurilor): *private*, *protected*, *public*, *package.*

Implicit metodele/variabilele au scopul *package*, sunt non-statice (aparțin fiecărei instanțe), şi non-finale(pot fi suprascrise în clasele derivate). Tendința este să folosim modificatorii impliciți.

Programarea obiectuală însă, indică "ascunderea" metodelor/câmpurilor şi expunerea doar a celor strict necesare. Constantele ar trebui marcate static final. De asemenea metodele/câmpurile dacă se poate ar trebui declarate final, şi chiar private. Dacă se poate chiar şi static (metodele statice sunt mai rapide).

Când discutăm despre drepturile de acces la membrii unei clase trebuie să abordăm acest subiect din 2 perspective:

A. Interiorul clasei sau, mai concret, metodele clasei. În cadrul metodelor unei clase există acces nerestrictiv la toți membrii, date sau funcții. De exemplu, în metodele clasei **Punct** se face referire la câmpurile x şi y. În mod asemănător s-ar fi putut referi (apela) şi metodele. De exemplu, am fi putut defini funcția **move** pe baza funcției **init** astfel:

```
class Punct{
    //. . .
    public void move(int dx, int dy) {
        init(x+dx, y+dy);
    }
    //. . .
}
```

Se observă că în interiorul clasei nu se foloseşte notaţia cu punct pentru a referi membrii, aceştia fiind pur şi simplu accesaţi prin numele lor. Când o metodă face referire la alţi membri ai clasei, de fapt sunt accesaţi membrii corespunzători ai obiectului receptor, indiferent care ar fi el. De exemplu, cand se apelează metoda **init** a obiectului referit de **p1**, are loc iniţializarea membrilor x şi y ai acelui obiect. În legatură cu accesul din interiorul unei clase, trebuie spus că absenţa restricţiilor se aplică şi dacă este vorba despre membrii altui obiect din aceeaşi clasă, dar diferit de cel receptor. De exemplu, dacă în clasa **Punct** am avea câte o metodă de calcul al distanţei pe vericală/orizontală dintre 2 puncte, unul fiind obiectul receptor, iar celălalt un obiect dat ca parametru, atunci am putea scrie:

```
class Punct{
    //. . .
    public int distV(Punct p) {
        return y - p.y;
    }
    public int distH(Punct p) {
        return x - p.x;
    }
    //. . .
}
```

Se observă că din interiorul metodelor **distV / distH** putem accesa liber membrii privaţi ai obiectului **p** dat ca parametru. La fel ar sta lucrurile şi daca **p** ar fi o variabilă locală a unei metode din clasa **Punct**.

B. Exteriorul sau clienţii clasei. Clienţii unei clase pot accesa doar acei membri care au ca modificator de acces cuvântul public. Membrii declaraţi cu modificatorul private NU sunt vizibili în afară, sunt ascunşi. Dacă am încerca să folosim în metoda **main** din exemplul nostru o referinţă de genul:

```
p1.x
```

compilatorul ar raporta o eroare. O observaţie importantă pe care o putem desprinde din exemplul clasei **Punct** este aceea că structura unei clase, sau modul ei de reprezentare, care este dat de variabilele membru, de regulă se ascunde faţă de clienţi. Dacă este necesar ca aceştia să poată consulta valorile datelor membru, se va opta pentru definirea unor metode de genul **getValoare**, iar nu pentru declararea datelor respective ca fiind publice.

Într-o clasă Java putem declara membri care să nu fie precedaţi de nici unul dintre modificatorii public sau private. În acest caz, membrii respectivi se spune că sunt accesibili la nivel de pachet ("package").

- Să considerăm următorul exemplu:

Un program pentru Pentagon ce controlează toate rachetele nucleare adăpostite în silozuri de pe teritoriul S.U.A. În acest caz, programul ar putea folosi obiecte de tip *silo* (vezi clasa *silo* de mai jos). Din motive de securitate, pentru a lansa o rachetă programul trebuie să specifice o parolă. Dacă parola este corectă, variabila *lanseaza_rachete* primeşte valoarea 1, iar rachetele vor fi lansate. Dacă parola nu este corectă, variabila rămâne 0 iar rachetele nu vor fi lansate. Dacă toţi membrii clasei ar fi declaraţi *public*, atunci orice clasă Java care utilizează obiecte *silo* va putea folosi următoarea instrucţiune pentru a lansa rachete ignorând funcţia de acces prin parolă (*void lansare_rachete(char *Parola)*):

```
wyoming_silo.lanseaza_rachete=1;
```

(unde *wyoming_silo* este un obiect de tipul *silo*)

Când se folosesc obiecte, accesul la majoritatea variabilelor membru trebuie limitat la funcțiile membru. Astfel, singura modalitate ca programul să aibă acces la variabila membru *lanseaza_rachete* este prin folosirea unei funcții membru (în cazul nostru *lansare_rachete(char *Parola)*), adică programul este forțat să joace după regulile impuse de creatorul său. Pentru a restricționa accesul la membrii clasei, se poate folosi cuvântul cheie *private*.

```
class silo
{
  private  String locatia;
  private int tip_racheta;
  private int lanseaza_rachete; // daca este 0 nu se lanseaza,
                                // daca este 1 se lanseaza
  private String parola;
  public  silo(int tipRacheta,  String loc)
  {
     tip_racheta=tipRacheta;
     locatia=loc;
     lanseaza_rachete=0;
     parola="hillary";
  }
  public void lansare_rachete(String Parola)
  {
   if (parola.compareTo(Parola)==0)
           lanseaza_rachete=1;
   else lanseaza_rachete=0;
  }


  public void mesaj()
  {
    if (lanseaza_rachete==1)
    {
     System.out.println("parola corecta; rachetele s-au lansat");
     System.out.println("tip rachete="+tip_racheta+" --
                         destinatie="+locatia+" \n");
    }
    else
     System.out.println("parola incorecta; rachetele nu s-au
                         lansat \n");
  }
}

public class siloTest
{
   public static void main(String args[ ])
   {

    silo s=new silo(1,"Moscova");
    s.lansare_rachete("ana");
    s.mesaj();
    s.lansare_rachete("hillary");
    s.mesaj();
   }
}
```

- Să considerăm şi alte câteva exemple utile:

```
package pac1;
public class produs
{
  private String denumire;
  protected int cantitate;
  public int cod;
  public  produs(String den,int cant,int cod)
  {
     denumire=den;
     cantitate=cant;
     this.cod=cod;
  }
  public void afis( )
  {
    System.out.println("denumire="+denumire+"
                        cantitate="+cantitate+" cod="+cod+" \n");
  }
}
```

```
package pac2;
import pac1.produs;
public class TestProdus
{
  public static void main(String args[ ])
  {
    produs p=new produs("Poiana",23,287);
    p.afis( );
    p.cod=112; // ok; "cod" este "public"
    p.afis();
//    p.cantitate+=10; eroare - "cantitate" este "protected"
//    p.denumire="Laura"; eroare - "denumire" este "private"
  }
}
```

```
package pac;
class produs
{
  private String denumire;
  protected int cantitate;
  public int cod;
  public  produs(String den,int cant,int cod)
  {
     denumire=den;
     cantitate=cant;
     this.cod=cod;
  }
  public void afis()
  {
    System.out.println("denumire="+denumire+"
                        cantitate="+cantitate+" cod="+cod+" \n");
  }
}
```

```
package pac;
public class TestProdus
{
  public static void main(String args[])
  {
    produs p=new produs("Poiana",23,287);
    p.afis();
    p.cod=112; // ok; "cod" este "public"
    p.cantitate+=10;// ok - "cantitate" este "protected"
    p.afis();
//    p.denumire="Laura"; eroare - "denumire" este "private"
  }
}
```

4. Clase abstracte. Interfețe în Java

Cateodată avem nevoie să declarăm o clasă, dar nu ştim cum să definim toate metodele care aparțin clasei. De exemplu, sa încercăm să declarăm o clasă numită **Mamifer** în care să includem o metoda membru numită **MarcheazaTeritoriul()**. Oricum, nu ştim cum să scriem **MarcheazaTeritoriul**() pentru ca aceasta se întâmplă diferit în funcție de specia de Mamifer . Bineînțeles, o să rezolvăm acest lucru derivând subclase din clasa **Mamifer**, clase cum ar fi **Maimuta** şi **Om**. Dar ce cod să conțină funcția **MarcheazaTeritoriul**() din clasa **Mamifer**?
În Java funcția **MarcheazaTeritoriul()** se poate declara în clasa **Mamifer** ca fiind o metodă **abstract**. Făcând aceasta se permite declararea metodei fără a scrie cod pentru subclase. Se va scrie însă cod în subclase.

Dacă o metodă este declarată **abstract,** atunci şi clasa trebuie declarată **abstract**. Pentru **Mamifer** şi subclasele sale, acest lucru înseamnă că trebuie să arate după cum urmează:

```
abstract class Mamifer {
      abstract void MarcheazaTeritoriul();
      }
public class Om extends Mamifer {
      public void MarcheazaTeritoriul() {
            //cod prin care se marcheaza teritoriul construind un gard
            }
      }

public class UnMembruAlGastii extends Mamifer {
      public void MarcheazaTeritoriul() {
      //cod prin care se marcheaza teritoriul cu graffiti
            }
      }

public class Maimuta extends Mamifer {
      public void MarcheazaTeritoriul() {
            //cod prin care se marcheaza teritoriul cu frunze si crengi
            }
      }
```

În declarațiile precedente, clasa **Mamifer** nu conține nici un fel de cod pentru **MarcheazaTeritoriul()**. Clasa **Om** poate conține cod despre cum omul marchează teritoriul construind un gard în jurul lui, în timp ce clasa **UnMembruAlGastii** poate conține cod despre cum acesta îşi marchează teritoriul pictând cu spray graffiti. Clasa **Maimuta** işi va marca teritoriul cu ajutorul crengilor (sau altfel).

Tipic, o clasă abstractă va avea câteva metode declarate abstract şi altele care nu sunt declarate astfel. Dacă se declară o clasă care este în totalitate abstractă, atunci se declară ceea ce în Java este cunoscut sub numele de **interfață**. O interfață este o clasă abstractă în întregime. Se pot deriva clase dintr-o interfață într-o manieră complet analoagă cu aceea a derivării claselor din alte clase.

Ca exemplu, să presupunem că dezvoltăm o aplicație care trebuie să afişeze ora. Utilizatorii vor avea două opțiuni pentru a obține această informație. Pot să o preia dintr-un ceas electronic sau dintr-un ceas cu limbi. S-ar putea implementa ca mai jos:

```
interface Ceas {
      public String CitesteTimpul(int ora);
      }

class CeasMecanic implements Ceas {
      public String CitesteTimpul(int ora) {
            StringBuffer str = new StringBuffer();
                  for (int i=0; i < ora; i++)
                        str.append("CeasCuLimbi ");
                        return str.toString();
                        }
      }

class CeasElectronic implements Ceas {
      public String CitesteTimpul(int ora) {
            return new String("Este ora " + hour + ":00");
            }
      }
```

In acest exemplu, **Ceas** este ***o interfață*** care furnizează o singură funcție, **CitesteTimpul()**. Aceasta înseamnă că orice clasă care este derivată (sau, în alte cuvinte, **implementată** - *implements*) din interfața Ceas trebuie să implementeze o funcție **CitesteTimpul()**. **CeasCuLimbi** este un exemplu de clasă care implementează **Ceas**, şi săi observăm că în loc de **class CeasCuLimbi extends Ceas**, sintaxa care ar fi trebuit folosită în cazul în care **Ceas** ar fi fost o clasă abstractă, este folosita sintaxa **class CeasCuLimbi implements Ceas**.

Pentru că **CeasCuLimbi** implementează interfața **Ceas**, ea furnizează o funcție **CitesteTimpul**(). Clasa **CeasElectronic** implementează de asemenea **Ceas** şi furnizează o funcție **CitesteTimpul**().

- ## Precizări:

1. Interfețele şi superclasele nu se exclud reciproc. O clasă nouă poate fi derivată dintr-o superclasă şi poate implementa una sau mai multe interfețe. Acest lucru se poate efectua ca mai jos, pentru o clasă care implementează doua interfețe şi are o superclasă:

```
class SubClasaMea extends SuperClasaMea implements
                           PrimaInterfata, ADouaInterfata
{      // implementarea clasei  }
```

Pentru că este posibil ca o clasă să implementeze mai mult de o interfață, interfețele reprezintă o alternativă pentru moştenirea multiplă, care nu este permisă în Java.

2. Metodele abstracte sunt metode care nu au corp de implementare. Ele sunt declarate

numai pentru a forța subclasele, care se vor instanția, să implementeze metodele respective.

3. Metodele abstracte trebuie declarate numai în interiorul claselor care au fost declarate abstracte. Altfel compilatorul va semnala o eroare de compilare. Orice subclasă a claselor abstracte care nu este declarată abstractă trebuie să ofere o implementare a acestor metode, altfel va fi generată o eroare de compilare.
4. Prin acest mecanism ne asigurăm că toate instanțele care pot fi convertite către clasa care conține definiția unei metode abstracte au implementată metoda respectivă dar, în acelaşi timp, nu este nevoie să implementăm în nici un fel metoda chiar în clasa care o declară, deoarece nu ştim pe moment cum va fi implementată.
5. O metodă statică nu poate fi declarată şi abstractă pentru că o metodă statică este implicit finală şi nu poate fi rescrisă.

III. MODUL DE LUCRU

- Clasic:

1. Se editează codul sursă al programului Java folosind un editor de text disponibil (de ex., se poate utiliza Notepad).
2. Se salvează fişierul cu extensia **.java**.
3. Compilarea aplicației Java se va face din linia de comandă:
   javac nume_fişier_sursă.java
   În cazul în care programul conține erori acestea vor fi semnalate şi afişate.
4. Pentru rularea aplicației Java, se lansează interpretorul Java:
   java nume_clasă_care_conține_main

- Se foloseşte utilitarul disponibil în laborator J2SDK Net Beans IDE.

IV. TEMĂ

- Se vor parcurge toate exemplele prezentate în platforma de laborator testându-se practic (acolo unde este cazul).

- Implementați clasa **Mamifer** din cadrul exemplului de mai sus astfel:
  - scrieți codul pentru implementarea metodei: public void **MarcheazaTeritoriul**(), în fiecare din clasele: **Om**, **UnMembruAlGastii**, **Maimuta**;
  - încercați să apelați metoda: public void **MarcheazaTeritoriul**(), în cadrul claselor, şi din afara lor.
  Explicați, la fiecare caz în parte, rezultatele obținute.

- Integrați clasele **Televizor** şi **TelevizorColor** într-un program Java funcțional, în care să se definească câte două instanțe pentru fiecare din clasele de mai sus şi să se folosească metodele definite pentru a afişa proprietățile instanțelor.
  Explicați la fiecare caz în parte rezultatele obținute.