TEHNOLOGII JAVA
PENTRU DEZVOLTAREA APLICAȚIILOR
LUCRARE DE LABORATOR 7

Variabile „shadow”. Suprascrierea metodelor in Java Ascunderea si încapsularea datelor.

I. SCOPUL LUCRĂRII

Lucrarea de față are rolul de a prezenta şi familiariza studentul cu noțiuni ale limbajului Java referitoare la: variabile “shadow”, suprascrierea metodelor, ascunderea şi încapsularea datelor.

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

II. NOȚIUNI TEORETICE

1. Variabile “shadow”

Vezi Curs 7, secțiunea 4.6.

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:

Observăm declarația în clasa **TelevizorColor** a unei variabile cu numele **diagonala**, care „umbreşte" variabila cu acelaşi nume declarată în clasa **Televizor**.

Dacă accesăm numele **diagonala** sau sinonimul **this.diagonala** în cadrul clasei **TelevizorColor** vom accesa variabila **diagonala** declarată în **TelevizorColor**.

Altfel, putem sa distribuim (cast) pe **this** clasei corespunzătoare apoi să accesăm **variabila shadow**: ***((Televizor this) .diagonala.***

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 in Java

Vezi Curs 7, secțiunea 4.7.

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:

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**.

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).

De asemenea, trebuie ştiut faptul că suprascrierea metodelor nu reprezintă, la nivelul metodelor, acelaşi lucru ca şi variabilele shadow (**vezi cursul 7, sectiunea 4.7**).

3. Ascunderea si incapsularea datelor

Vezi Curs 7, secțiunea 4.8.

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

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 notația cu punct pentru a referi membrii, aceştia fiind pur şi simplu accesați prin numele lor. Când o metodă face referire la alți membri ai clasei, de fapt sunt accesați membrii corespunzători ai obiectului receptor, indiferent care ar fi el. De exemplu, cand se apelează metoda **init** a obiectului referit de **p1**, are loc inițializarea membrilor x şi y ai acelui obiect. În legatură cu accesul din interiorul unei clase, trebuie spus că absența restricțiilor 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 distanței 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 privați 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 clienții clasei. Clienții unei clase pot accesa doar acei membri care au ca modificator de acces cuvântul public. Membrii declarați cu modificatorul private NU sunt vizibili în afară, sunt ascunşi. Dacă am încerca să folosim în metoda **main** din exemplul nostru o referință de genul:

```
p1.x
```

compilatorul ar raporta o eroare. O observație 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 față de clienți. 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 precedați 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 toți membrii clasei ar fi declarați *public*, atunci orice clasă Java care utilizează obiecte *silo* va putea folosi următoarea instrucțiune pentru a lansa rachete ignorând funcția 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");
  }
}
```

Fişierul **produs.java** se va salva în subdirectorul pac1 din:
c:\JBulider7\jdk1.3.1\bin\pac1

```
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"
  }
}
```

Fişierul **TestProdus.java** se va salva în subdirectorul pac2 din:
c:\JBulider7\jdk1.3.1\bin\pac2
Pentru lansarea în execuție se va utiliza comanda: **java pac2.TestProdus**

```
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");
  }
}
```

Fişierul **produs.java** se va salva în subdirectorul pac din:
c:\JBulider7\jdk1.3.1\bin\pac

```
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"
  }
}
```

Fişierul **TestProdus.java** se va salva în subdirectorul pac din:
c:\JBulider7\jdk1.3.1\bin\pac
Pentru lansarea în execuție se va utiliza comanda: **java pac.TestProdus**

III. MODUL DE LUCRU

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**. Fişierul trebuie salvat la următoarea locație: **c:\JBulider7\jdk1.3.1\bin**
3. Compilarea mini-aplicației Java se va face din linia de comandă. Se poate proceda astfel. Se deschide o fereastră MS-Dos: Start ->Run, se tipăreşte **command** în căsuța de text şi se apasă butonul OK. Printr-o schimbare de directoare şi subdirectoare se trece la locația: **c:\JBulider7\jdk1.3.1\bin.** Sau, se lansează WindowsCommander. Se trece la locația **c:\JBulider7\jdk1.3.1\bin.** Se deschide o fereastră MS-Dos: Commander ->Run Dos.
4. Pentru compilare, se tipăreşte la prompter **javac nume_fişier_sursă.java** şi se apasă Enter. De ex., dacă fişierul se numeşte Test.java, se va scrie **javac Test.java**. În cazul în care programul conține erori acestea vor fi semnalate şi afişate în fereastra MS-Dos, după care va apare iar prompter-ul. Dacă programul nu conține erori şi compilarea se face cu succes, atunci va apare numai prompter-ul.
5. Pentru rularea aplicației Java, se lansează interpretorul Java. Se tipăreşte la prompter următoarea comandă **java nume_clasă_care_conține_main** şi se apasă Enter. De ex., dacă clasa din program care conține metoda *main( )* se numeşte Test, se va scrie **java Test**.

IV. TEMĂ

1. Se vor parcurge toate exemplele prezentate în platforma de laborator testându-se practic, explicând rezultatele obținute.
2. 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.