CAPITOLUL 2

2.1. Moştenirea (Inheritance)

Numită și **derivare**, moștenirea este un mecanism de reutilizare a codului specific limbajelor orientate obiect și reprezintă posibilitatea de a defini o clasă care **extinde** o altă clasă deja existentă. Ideea de bază este de a **prelua** funcționalitatea existentă într-o clasă și de a **adăuga** una nouă sau de a o **modela** pe cea existentă.

Clasa existentă este numită **clasa-părinte**, **clasa de bază** sau **super-clasă**.

Clasa care extinde clasa-părinte se numește **clasa-copil (child)**, **clasa derivată** sau **sub-clasă**.

Moștenirea este un mecanism care permite crearea unor versiuni "specializate" ale unor clase existente (de bază). Moștenirea este folosită în general atunci când se dorește construirea unui tip de date care să reprezinte o implementare specifică (o specializare oferită prin clasa derivată) a unui lucru mai general.

O clasă poate moşteni variabilele şi metode de la altă clasă, folosind cuvântul cheie **extends**. Subclasa are acces la toate variabilele publice şi a metodelor din clasa părinte. În acelaşi timp, o subclasă poate avea metode cu acelaşi nume şi parametri ca metodele de superclasa. În acest caz, subclasa suprascrie metodele clasei părinte. Următorul exemplu suprascrie metoda **show()**, care este prezentă în două clase **Bird** şi **Eagle**. În conformitate cu principiul polimorfismului se apelează metoda, care este cea mai apropiata de obiectul curent.

```
    /* exemplu # 1 : moştenirea clasei şi
suprascrierea metodei : */
class Bird {
     private float price;
     private String name;
```

```
public Bird(float p, String str) {
//constructorul
     price = p;
     name = str;
     }
public float getPrice(){
     return price;
     }
public String getName(){
     return name;
     }
void show(){
System.out.println("denumirea:" + name +   "
, preţul: "+ price);
     }}
class Eagle extends Bird {
    private boolean fly;
public Eagle(float p,String str,boolean f) {
 super(p, str);
 //apelul constructorului superclasei
 fly = f;
}
void show(){
System.out.println("denumire:" + getName() +
", preţ: " + getPrice() + ", zbor:" + fly);
   }}
public class BirdSample {
    public static void main(String[] args) {
Bird b1 = new Bird(0.85F, "Gâscă");
Bird b2 = new Eagle(10.55F, "Vultur", true);
b1.show(); // apelul show() a clasei Bird
 b2.show(); // apelul show()a clasei Eagle
  }}
```

Obiectul b1 este creat prin apelarea constructorului clasei **Bird**, şi, desigur, prin metoda **show()**, se apelează versiunea metodei din clasa **Bird**. La crearea obiectului **b2** link-ul de tipul

Bird este iniţializat de tipul **Eagle**. Pentru aşa metodă de iniţializare a link-ul superclasa primeşte acces la metodele, suprascrise în subclasă.

La declararea câmpurilor care coincid după nume în superclasa şi subclasa, valorile lor nu se redefinesc şi nu se intersectează, adică exista obiecte cu acela nume care nu depind unul de altul. În acest caz, problema de a extrage valoarea cerută de anumite domenii, care aparţin clasei în lanţul de moştenire, depinde de programator.

```
    /* exemplu # 2 : accesul la câmpuri cu
acela nume la moştenire: */
class A {
 int x = 1, y = 2;
    public A() {
        y = getX();
    System.out.println("în clasa A după
apelarea metodei getX() x="+x+"  y="+y);
    }
public int getX(){
     System.out.println("În clasa A");
    return x;
     } }
class B extends A {
 int x = 3, y = 4;
    public B() {System.out.println
("În clasa B x=" + x + "  y=" + y);
    }
public int getX(){
     System.out.println("În clasa B");
    return x;
     } }
public class DemoAB {
   public static void main (String[] args) {
        A objA = new B();
        B objB = new B();
System.out.println(objA.x);
```

```
System.out.println(objB.x);
      }}
```

La realizare va fi:

```
În clasa B
În clasa A după apelul metodei getX()  x=1  y=0
În clasa B  x=3  y=4
În clasa B
În clasa A după apelul metodei getX()  x=1  y=0
În clasa B  x=3  y=4
x=1
x=3
```

In cazul creării obiectului **objA** iniţializând link-ul clasei **A** cu obiectului clasei **B**, a fost primit accesul la câmpul **x** a clasei **A**. În al doilea caz, la crearea unui obiect **objB** a clasei B a fost primit accesul la câmpul **x** a Clasei **B**. Cu toate acestea, folosind un tip de conversie de forma: **((A) objB).x** sau **((B) objA).x** poate fi accesat cu uşurinţă câmpul **x** din clasa corespunzătoare.

O parte din tehnicile polimorfismului ilustrează constructorul din clasa **A** în forma:

```
public A() {
   y = getX();
        }
```

Metoda **getX()** este în clasa **A**, cit şi în clasa **B**. La crearea unui obiect a clasei **B** cu una din metodele :

```
A objA = new B();
B objB = new B();
```

În orice caz, primul se apelează constructorul din clasa **A**, dar aşa cum se creează un obiect a clasa B, şi metoda **getX()** respectiv se apelează din clasa **B**, care la rândul său, operează cu cîmpul **x** care nu este încă iniţializat pentru clasa **B**. Ca rezultat **y** va primi valoarea **x** în mod implicit, care este zero.

Nu se poate crea o subclasă pentru clasa, declarată cu calificativul **final**:

```
// clasa First nu poate fi superclasă
final class First {
```

```
/*codul programului */}
// următoarea clasă nu este posibilă
class Second extends First{
    /*codul programului */}
```

2.2. Utilizarea super şi this

Cuvântul cheie **super** este folosit pentru a activa constructorul superclasei şi pentru accesul la membrii din superclasa. De exemplu:

```
     super(lista parametrilor);
     /*   activarea   constructorului   din
 superclasă cu transmiterea parametrilor */
super.i = n;
//atribuirea valorii obiectului superclasei
super.method();
// activarea metodei din superclasă
```

A doua formă **super** este ca un link **this** la exemplarul clasei. A treia formă este specifică pentru Java şi oferă un apel la metoda suprascrisei iar în cazul în care in această superclasă metoda nu este definită, atunci se va căuta în lanţul de moştenire atâta timp pînă cînd metoda nu va fi găsită. Fiecare exemplu de clasă are implicit link-ul **this** la sine, care se transmite şi la metode. Putem scrie **this.price** dar nu este neapărat.

Următorul cod arata, cum, se foloseşte **this**, pentru a construi constructori bazaţi pe alţi constructori.

```
     // exemplu # 3 : this în constructori :
class Locate3D {
  private int x, y, z;
 public Locate3D(int x, int y, int z) {
   this.x = x;
   this.y = y;
   this.z = z;
          }
 public Locate3D() {
     this(-1, -1, -1);}}
```

În această clasă, al doilea constructor este pentru a finaliza iniţializarea unui obiect ce se referă la primul constructor. Această construcţie este utilizată in cazul în care o clasa are mai mulţi constructori şi apare necesitatea de a adăuga un constructor implicit.

Link-ul **this** este utilizat în metodă pentru a concretiza despre care variabile **x** şi **y** merge vorba de fiecare dată, şi în special pentru a accesa variabila clasei, daca in metoda este o variabilă cu acelaşi nume. Instrucţiunea **this** ar trebui să fie unica in constructorul apelat şi sa fie prima operaţie efectuata.

Lucrare de laborator nr. 2

1. Tema lucrării:
Moştenirea
.

2. Scopul lucrării:

- Însuşirea principiilor de moştenire a claselor;

3. Etapele de realizare:

1) Crearea subclacelor;
2) Apelarea metodelor superclasei în subclase;
3) Utilizarea obiectelor la moştenire;
4) Prezentarea lucrării.

4. Exemplu de realizare:

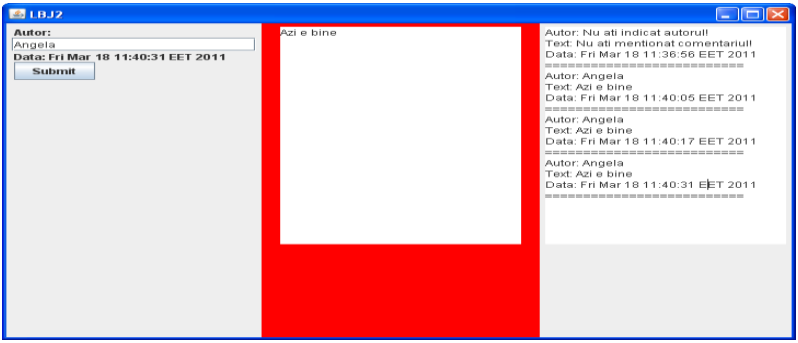
```
import java.awt.*;
import java.awt.event.*;
import java.lang.annotation.*;
import java.util.*;
import javax.annotation.*;
import javax.swing.*;
public class Lb3 {
    public static void main(String[] arg){
    MyFrame ff = new MyFrame();
    ff.setVisible(true);
    }}
```

```
class MyAnnotation implements Generated {
    public MyAnnotation(){
        this.setAnnotation();
    }
     public MyAnnotation(String cCom,
     String uUser){
        this.setAnnotation(cCom, uUser);
    }
    public void setAnnotation(){
      com = "Nu aţi menţionat comentariul!";
        user = "Nu aţi indicat autorul!";
        dat = new Date();
    }
    public void setAnnotation(
     String cCom,String uUser){
        com = cCom;
        user = uUser;
        dat = new Date();
    }
    public String date() {
        return dat.toString();
    }
    public String comments() {
        return com;
    }
    public String users(){
        return user;
    }
    public String getAnnotation(){
        String output;
        output = "Autor: " + user + "\n";
        output += "Text: " + com + "\n";
        output += "Data: " + date() + "\n";
   output += "=========================\n";
        return output;
    }
```

```
    private String com;
    private Date dat;
    private String user;
    public Class <? extends Annotation>
annotationType() {
        throw new
UnsupportedOperationException(
     "Not supported yet.");
    }
     public String[] value() {
        throw new
UnsupportedOperationException("
     Not supported yet.");
    }}
class MyFrame extends JFrame {
    public MyFrame(){
        this.setTitle("LBJ2");
        this.setSize(600, 500);
        this.setLocation(300, 300);
this.setDefaultCloseOperation(
JFrame.EXIT_ON_CLOSE);
        ts = new MyAnnotation();
       user = new JTextField(ts.users(),10);
        left = new JPanel();
        Box b1 = Box.createVerticalBox();
        JButton but = new JButton("Submit");
        but.addActionListener(
     new ActionListener(){
 public void actionPerformed(ActionEvent e){
                System.out.println("aa");
ts.setAnnotation(com.getText(),
user.getText());
          dat.setText("Data: "+ts.date());
history.setText(history.getText()+
ts.getAnnotation());
            }  });
```

```
        user = new JTextField(ts.users(),5);
      dat = new JLabel("Data: "+ts.date());
        b1.add(new JLabel("Autor:"));
        b1.add(user);
        b1.add(dat);
        b1.add(but);
        left.add(b1);
        this.add(left,BorderLayout.WEST);
        center = new JPanel();
        com = new
JTextArea(ts.comments(),20,20);
        center.add(com);
        center.setBackground(Color.red);
this.add(center,BorderLayout.CENTER);
        right = new JPanel();
        history = new
JTextArea(ts.getAnnotation(),20,20);
        right.add(history);
        this.add(right,BorderLayout.EAST);
    }
    private JTextArea history;
    private JPanel left, center, right;
    private JTextField user;
    private JTextArea com;
    private JLabel dat;
    private MyAnnotation ts;
}
```

Rezultatul realizării programului:



5. Sarcina spre realizare:

Scrieți un cod care ar conține două clase legate prin moștenire acestea să fie funcționale și să satisfacă următoarele cerințe:

- Clasa de bază să conțină 3 constructori și 2 metode
- Clasa dirivată să conțină 2 constructori și 3 metode
- Constructorii și metodele să fie apelați cu cuvintele cheie **this** și **super**
- Creați următoarele obiecte în metoda main():
    a) Obiectul clase de bază folosind constructorul clasei de bază;
    b) Obiectul clase derivate folosind constructorul clasei de bază;
    c) Obiectul clase de derivate folosind constructorul clasei derivate;
    d) Obiectul clase de bază folosind constructorul clasei derivate;
- Metodele declarate să fie apelate utilizând toate obiectele existente;