# Documentație pentru laboratorul 2

* 1. **Clase, obiecte, constructori.**

**1.1.1. Clase**

Clasele definesc structura şi comportamentul unui set de elemente din domeniul pentru care este dezvoltat software-ul modelului. Într-un obiect orientat sunt patru tipuri de relatii: dependenţă, generalizare, asociere, şi realizarea. Dependenţa este raportul de utilizare, indicând faptul că schimbarea starii obiectului al unei clase poate afecta obiectul altei clase pe care o utilizează. Generalizarea înseamnă că obiectele clasei derivate pot fi folosite ori de câte ori există obiecte a clasei de bază dar nu invers. Realizarea este raportul dintre clasificatori (clase, interfeţe), în care unul dintre ei descrie un alt clasificator. Asociaţia arată că obiectele dintr-o clasă se referă la obiecte din altă clasă, şi să reflectă între ele. Tipul "class" în Java poate conţine câmpuri de date, constructori, metode, clase interne şi interfeţe.

**1.1.2. Constructori**

Constructorul este o metodă care se activează în mod automat atunci când se crează o clasă şi poate efectua acţiuni asupra obiectului pe care îl iniţializează. Constructorul are acelaş nume ca si clasa. Constructorul nu returnează valori, dar poate avea parametri pentru a fi supraîncărcat.

/\*Exemplu 1 de supraîncarcare a constructorului: \*/

**class** NewBook {

**private** String title, publisher;

**private** **float** price;

**public** NewBook() {

title = “NoTitle”;

}

**public** NewBook(String t,String pub,**float** p) {

title = **new** String(t);

publisher = pub;

price = p;

}

**public** **static** **void** main(String[] args) {

NewBook tips1;

tips1 = **new** NewBook();

// initializarea

NewBook tips2 =

**new** NewBook("Java2", "C++", 9.f);

}}

În cazul în care constructorul nu este definit în clasa, Java oferă un constructor implicit care iniţializează obiectul cu valori implicite.

**1.1.3. Clase-Interfeţe**

Pentru a nu pune în aplicare toate metodele din interfeţele corespunzătoare, atunci când se creează clasa - blocuri pentru evenimente, sunt folosite clase – Interfeţe. Aceste clase contin metode interfeţe, evenimente realizate pe care le extinde. În acest caz, se defineşte o nouă clasă care acţionează ca un bloc de evenimente realizate şi realizează doar evenimentele care necesită. De exemplu, clasa **MouseMotionAdapter** are două metode**: mouseDragged()** şi **mouseMoved(**). Esenta acestor metode sunt aceleaşi ca şi în interfaţa **MouseMotionListener**. Dacă există necesitatea doar în evenimentele de clic a mouse-ului, se extinde adaptorul realizînd override cu metoda **mouseDragged()** din această clasa. Evenimentul de clic a mouse-ului se realizeaza prin aplicarea metodei **mouseMove().** Pentru a crea o interfaţă grafică trebui să fie indicat un loc (fereastră), în care aceasta va fi afişată. Fereastra este un container care creaza o interfaţă cu utilizatorul. Contextul grafic este incapsulat în clasă şi este disponibil în două moduri:

• prin activarea metodelor **paint(), update();**

• prin valoarea returnata a metodei **getGraphics()** din clasa **Component.**

Evenimentele **FocusEvent** indică că componentele au fost înregistrate sau nu. Clasa **InputEvent** este clasa de bază pentru clasele **KeyEvent** **şi MouseEvent**. Evenimentul **WindowEvent** indică un program in care a fost activată una din sistemele de control a fereastrei. Următorul exemplu creează obiectul **MyMouseWithFrame** şi permite controlul său direct in metoda **main()**.

/ / Exemplu # 2 : utilizarea interfeţii:

**import** java.applet.\*;

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** MyMouseWithFrame **extends** Frame **implements** ActionListener{

Button button = **new** Button("Button");

**public** String msg = "none";

**public** MyMouseWithFrame() {

addMouseListener(**new** MyMouseAdapter(**this**));

setLayout(**null**);

setBackground(**new** Color(255, 255, 255));

setForeground(**new** Color(0, 0, 255));

button.setBounds(100, 100, 50, 20);

button.addActionListener(**this**);

add(button); }

**public** **static** **void** main(String[] args) {

MyMouseWithFrame myf=**new** MyMouseWithFrame();

myf.addWindowListener(**new** WindowAdapter() {

**public** **void** windowClosing(WindowEvent e) {

System.exit(0);

};});

myf.setSize(**new** Dimension(250, 200));

myf.setTitle("Frame - window");

myf.setVisible(**true**);

}

**public** **void** paint(Graphics g) {

g.drawString(msg, 80, 50);

}

**public** **void** actionPerformed(ActionEvent e) {

msg = "Button is pressed";

repaint();

}}

**class** MyMouseAdapter **extends** MouseAdapter {

**public** MyMouseWithFrame mym;

**public** MyMouseAdapter(MyMouseWithFrame mym) {

**this**.mym = mym;

}

**public** **void** mousePressed(MouseEvent me) {

mym.msg = "Mouse button is pressed";

mym.repaint();

}

**public** **void** mouseReleased(MouseEvent me){

mym.msg = "Mouse button is released";

mym.repaint();}}

Rezultatul realizării programului:

Constructorul clasei foloseste **MyMouseWithFrame addMouseListener (new MyMouseAdapter (this))** pentru a înregistra evenimentele **mouse**. La crearea unui obiect de tip **MyMouseWithFrame**, metoda dată indică ca obiectul este interesat în prelucrarea anumitor evenimente. Clasa abstracta **MouseAdapter** este utilizata pentru a trata evenimente asociate cu **mouse** prin crearea blocului care conţine metodele **mousePressed(MouseEvent e), mouseReleased(MouseEvent e)**. Când un obiect generează evenimentul **WindowEvent**, obiectul **MyMouseWithFrame** analizează dacă este un eveniment **WindowClosing**. Dacă nu, obiectul **MyMouseWithFrame** îl ignorează. Dacă este evenimentul aşteptat, programul incepe procesul de realizare a activităţii sale. Clasa abstractă **WindowAdapter** este utilizată pentru primirea şi prelucrarea evenimentului. Clasa conţine metodele reîncărcare **windowActivated (WindowEvent e)**, apelată la activarea ferestrei şi **windowSlosing (WindowEvent e)** apelată la închiderea ferestrei.

**2.1. Supraîncărcarea metodelor**

Capacitatea limbajului Java de a selecta metoda necesară rezumând din tipul run-time se numeşte polimorfism dinamic. Căutarea metodei are loc întâi in clasa curentă, apoi în superclasă, până cînd metoda nu va fi găsită sau nu va ajunge la **Object** - superclasă pentru toate clasele.

Metodele statice pot fi reîncărcate într-o subclasă, dar nu pot fi polimorfice, deoarece apelul lor nu afectează obiecte.

Dacă două metode cu nume identice sunt în aceeaşi clasă, listele de parametri trebuie sa fie diferit. Aceste metode sunt supraîncărcate (overload). Daca metoda din subclasa coincide cu metoda din superclasă, atunci metoda subclasei este suprascrisă (override) metoda din superclasă. Toate metodele din Java sunt virtuale (cuvântul cheie virtual ca în C + + nu este folosit).

In următorul exemplu are loc aplicarea polimorfismului bazat pe dinamica înlănţuită. Deoarece superclasa conţine metode care sunt suprascrise în subclase, atunci obiectul superclasei va apela metodele diferitor subclase, în dependenţă de faptul la care obiect a subclasei are referinţa.

/\*Exemplu#1:legaturile dinamice a metodelor \*/

**class** A {

**int** i, j;

**public** A(**int** a, **int** b) {

i = a;

j = b;

}

**void** show() {

System.out.println("i şi j:"+ i +" " + j);}}

**class** B **extends** A {

**int** k;

**public** B(**int** a, **int** b, **int** c) {

**super**(a, b);

k = c;

}

**void** show() {

/\*metoda suprascrisă show() din A \*/

**super**.show(); //extragerea valorilor din A

System.out.println("k: " + k);

}}

**class** C **extends** B {

**int** m;

**public** C(**int** a, **int** b, **int** c, **int** d) {

**super**(a, b, c);

m = d;

}

**void** show() {

/\* metoda suprascrisă show() din B \*/

**super**.show(); //extragerea valorilor din B

// show();

/\*imposibil!! Metoda se va apela pe sine, ce va aduce la error în timpul realizării \*/

System.out.println("m: " + m);

}}

**public** **class** DynDispatch {

**public** **static** **void** main(String[] args) {

A Aob;

B Bob = **new** B(1, 2, 3);

C Cob = **new** C(5, 6, 7, 8);

Aob = Bob; // instalarea referinţei la Bob

Aob.show(); // apelul show() din B

System.out.println();

Aob = Cob; // instalarea referinţei la Cob

Aob.show(); // apelul show() din C

}}

Rezultatul realizării:

i şi j: 1 2

k: 3

i şi j : 5 6

k:7

m: 8

La apelarea show () tratarea **super** întotdeauna merge la cea mai apropiată superclasă.

**2.2. Supraîncărcarea Metodelor**

Metoda se numeşte supraîncărcată în cazul în care există mai multe versiuni cu acelaşi nume, dar cu un set diferit de parametri. Supraîncărcarea poate fi limitată la o clasă sau mai multe clase, dar în mod necesar situată în acelaşi lanţ de moştenire. Metodele statice pot fi supraîncărcate cu nonstatice şi invers.

La apelarea metodelor de supraîncărcare ar trebui să evite situaţiile în care compilatorul nu va putea să aleagă o metodă sau alta, cum ar fi în cazul:

/\* exemplu # 2 : apelarea metodei supraîncărcate: \*/

**class** ClassC {}

**class** ClassD **extends** ClassC{}

**public** **class** DemoCD {

**static** **void** show(ClassC obj1, ClassD obj2){

System.out.println("prima metodă show(ClassC, ClassD)");

}

**static** **void** show(ClassD obj1, ClassC obj2){

System.out.println("a 2 metodă show(ClassD, ClassC)");

}

**static** **void** show(Object obj1, Object obj2){

System.out.println("a 3 metodă show(Object, Object)");

}

**public** **static** **void** main(String[] args) {

ClassC c = **new** ClassC();

ClassD d = **new** ClassD();

Object ob= **new** Object();

show(c,d);//1- prima metodă

show(d,c);//2-a doua metodă

show(c,c);//3-a treia metodă

//show(d,d);//4\_ errore

show(ob, ob);//5\_a treia metodă

show(c,ob);//6\_a treia metodă

show(ob,d);//7\_a treia metodă

}}

În primul caz, al 2-a şi al 5-a, parametrii in metoda **show()** complect coincid cu parametrii din declaraţia metodei. În al 3-a caz, prima şi a doua metodă nu pot fi utilizare, deoarece unul dintre parametrii acestei metode este un obiect al clasei **ClassD**, însă definirea metodei apelate merge in vârful lanţului de moştenire pentru parametrii, deci în acest caz, va fi apelată metoda cu parametrii de tipul **Object.** O situaţie similară apare în cazurile 6-a şi 7-a. În al 4-a caz primele doua metode show (), sunt la fel de potrivite pentru apel (analogic şi al 3-a), din acest motiv apare o eroare de compilare. Pentru a evita incertitudinea, trebuie de utilizat conversia de tip, cum ar fi:

**show(d,(ClassC)d);** **show(d,(Object)d);**

Fiecare dintre opţiuni in cele din urma vor apela metoda corespunzătoare show().

In următorul exemplu cream un exemplar a unei subclase cu ajutorul **new,** link-ul ei este transmis unui obiect a superclasei. La apelarea din superclasă, respectiv, se apelează metoda subclasei.

/\* exemplu # 3 : legăturile dinamice a metodelor: \*/

**class** A {

**void** myMethod() {/\* private nu se poate de folosit, pentru ca metoda la moştenire nu mai este disponibilă \*/

System.out.println("metoda clasei A");

}

**void** myMethod(**int** i) { System.out.println ("metoda clasei A cu argument");

}}

**class** B **extends** A {

**void** myMethod(**int** i) { System.out.println ("metoda clasei B cu argument ");

}}

**public** **class** C **extends** B {

{

System.out.println("clasa C");

}

**void** myMethod() {

System.out.println("metoda clasei C ");

}}

**public** **class** Dispatch {

**public** **static** **void** main(String[] args) {

A obj1 = **new** B();

obj1.myMethod();

A obj2 = **new** C();

obj2.myMethod();

obj2.myMethod(10);

}}

La realizare are loc:

metoda clasei A

clasa C

metoda clasei C

metoda clasei B cu argument

In primul rând se apelează metoda **myMethod()** din clasa **A**, ca moştenit. În al doilea rând se apelează metoda **myMethod()** din clasa **C**, ca redefinit. La urmă se apelează metoda **myMethod(int i)** din clasa **B**, astfel încât nu poate fi apelată metoda fără argument şi se îndeplineşte căutarea metodei necesare pe nivele de moştenire.

**Lucrare de laborator nr. 2**

**1. Tema lucrării:**

Supraîncărcarea metodelor.

**2.**  **Scopul lucrării:**

* Însuşirea principiilor de supraîncărcare a metodelor şi realizarea lor;
* Însuşirea principiilor de suprascriere a metodelor şi realizarea lor;

**3. Etapele de realizare:**

1. Crearea metodelor supraîncărcate;
2. Crearea metodelor suprascrise;
3. Utilizarea metodelor supraîncărcate în superclase şi subclase;
4. Utilizarea metodelor suprascrise;
5. Crearea interfeţii programului;
6. Prezentarea lucrării.

**4. Exemplu de realizare:**

import java.util.\*;

public class Prog2 {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

System.out.println("Al doilea program");

int n;

System.out.println("Nr de propoziţii");

n = in.nextInt();

Parag par = new Parag(n);

System.out.println(par);

}}

class Propoz {

public Propoz() {

prop = null;

}

public Propoz(String prop) {

this.prop = prop;

}

public Propoz(String prop, int n) {

this.prop = prop.substring(0,n-1);

}

public void setProp() {

prop = null;

}

public void setProp(String prop) {

this.prop = prop;

}

public void setProp(String prop, int n) {

this.prop = prop.substring(0,n);

}

public String getProp() {

return prop;

}

public String toString() {

String output = super.toString();

output += "(prop=\""+prop+"\")";

return output;

}

private String prop;

}

class Parag {

public Parag() {

par = null;

numProp = 0;

prop = null;

}

public Parag(int numProp) {

Scanner in = new Scanner(System.in);

this.numProp = numProp;

prop = new Propoz[this.numProp];

par = "";

for (int i=0;i<this.numProp;i++)

{

System.out.println("Daţi propoziţia "+(i+1)+":");

String s = in.nextLine();

par += s+" ";

System.out.println("Numărul de caractere");

int n = in.nextInt();

in.nextLine();

if (s.length()==0)

{

prop[i] = new Propoz();

}

else

{

if (n==0){

prop[i] = new Propoz(s);

}

else

{

prop[i] = new Propoz(s,n);

}}}}

public String toString() {

String output = super.toString();

output += "(";

for(Propoz t:prop)

{

output += t + ",";

}

output += par;

output += ")";

return output;

}

private Propoz[] prop;

private String par;

private int numProp;

}

**Rezultatul realizării**

Nr. de propoziţii

3

Daţi propoziţia 1:

Texte texte

Numarul de caractere

4

Daţi propoziţia 2:

Numărul de caractere

0

Daţi propoziţia 3:

Parametri noi

Numărul de caractere

9

**5. Probleme propuse:**

Să se realizeze supraîncărcarea şi suprascrierea a trei metode (utilizarea unei clase cu posibilităţile altei clase).

1. Sa se creeze obiectul clasa «Linie», folosind clasa «Cuvint».

2. Sa se creeze obiectul clasa «Paragraf», folosind clasa «Linie».

3. Sa se creeze obiectul clasa «Pagina», folosind clasa «Paragraf».

4. Sa se creeze obiectul clasa «Text», folosind clasa «Pagina».

5. Sa se creeze obiectul clasa «Paragraf», folosind clasa «Cuvint».

6. Sa se creeze obiectul clasa «Pagina», folosind clasa «Cuvint».

7. Sa se creeze obiectul clasa «Pagina», folosind clasa «Linie».

8. Sa se creeze obiectul clasa «Text», folosind clasa «Paragraf».

9. Sa se creeze obiectul clasa «Carte», folosind clasa «Text».

10. Sa se creeze obiectul clasa «Biblioteca», folosind clasa «Carte».

11. Sa se creeze obiectul clasa «Cuvint», folosind clasa «Litere».

12. Sa se creeze obiectul clasa «Litere», folosind clasa «Simboluri».