

Fluxuri

Programare Orientată pe Obiecte



Fluxuri

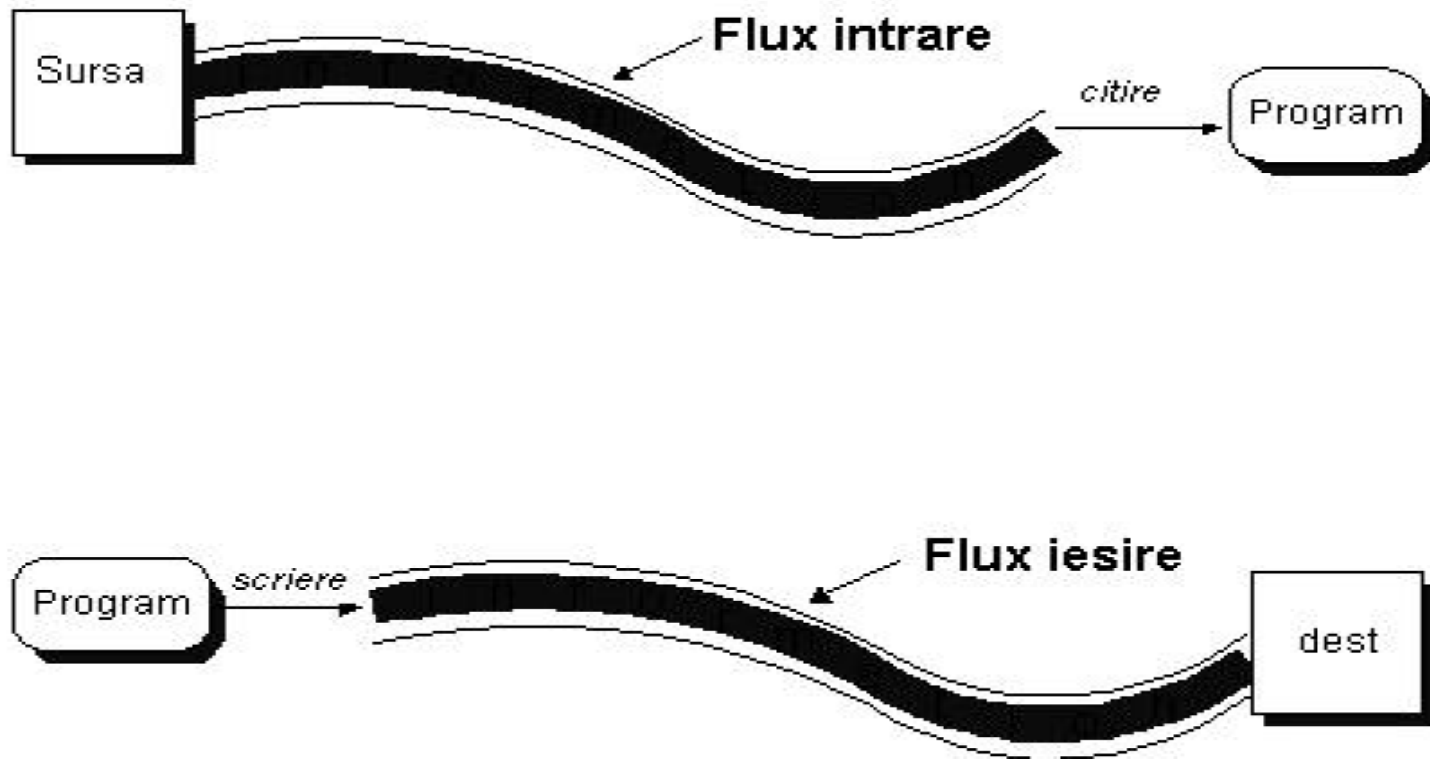


- Ce sunt fluxurile ?
- Clasificare, ierarhie
- Fluxuri primitive
- Fluxuri de procesare
- Intrări și ieșiri formatate
- Fluxuri standard de intrare și ieșire
- Analiza lexicală
- Clase independente
(RandomAccessFile, File)

Ce sunt fluxurile? (1)

- Schimb de informații cu mediul extern
- Canal de comunicație între două procese
- Seriale, pe 8 sau 16 biți
- Producător: proces care descrie o sursă externă de date
- Consumator: proces care descrie o destinație externă pentru date
- Unidirecționale, de la producător la consumator
- Un singur proces producător
- Un singur proces consumator

Ce sunt fluxurile? (2)



Un flux care citește date se numește flux de intrare.

Un flux care scrie date se numește flux de ieșire.

Ce sunt fluxurile? (3)

- Pachetul care oferă suport pentru operațiile de intrare/ieșire: `java.io`
- Schema generală de utilizare a fluxurilor:

```
deschide canal comunicatie  
while (mai sunt informatii) {  
    citeste/scrie informatie;  
}  
inchide canal comunicatie;
```

Clasificarea fluxurilor



- După direcția canalului de comunicație:
 - fluxuri de intrare (citire)
 - fluxuri de ieșire (scriere)
- După tipul de date:
 - fluxuri de octeți (8 biți)
 - fluxuri de caractere (16 biți)
- După acțiunea lor:
 - fluxuri primare (se ocupă efectiv cu citirea/scrierea datelor)
 - fluxuri pentru procesare

Ierarhia claselor pentru lucrul cu fluxuri

Caractere:

- Reader
 - FileReader, BufferedReader,...
- Writer
 - FileWriter, BufferedWriter,...

Octeți:

- InputStream
 - FileInputStream, BufferedInputStream..
- OutputStream
 - FileOutputStream,
BufferedOutputStream..

Metode comune fluxurilor

- Metodele comune sunt definite în superclasele abstracte corespunzătoare:

Reader	Writer
int read()	void write()
int read(char buf[])	void write(char buf[])
...	...

InputStream	OutputStream
int read()	void write()
int read(byte buf[])	void write(byte buf[])
	void write(String str)

- Inchiderea oricărui flux: close.
- Excepții: IOException sau derivate.

Fluxuri primitive

In funcție de tipul sursei datelor:

- Fișier
 - FileReader, FileWriter,
 - FileInputStream, FileOutputStream
- Memorie
 - CharArrayReader, CharArrayWriter,
 - ByteArrayInputStream,
 - ByteArrayOutputStream,
 - StringReader, StringWriter
- Pipe
 - PipedReader, PipedWriter,
 - PipedInputStream, PipedOutputStream
 - folosite pentru a canaliza ieșirea unui program sau fir de execuție către intrarea altui program sau fir de execuție.

Crearea unui flux primitiv

```
FluxPrimitiv numeFlux = new FluxPrimitiv  
    (dispozitivExtern);
```

- **crearea unui flux de intrare pe caractere**

```
FileReader in = new FileReader("fisier.txt");
```

- **crearea unui flux de iesire pe caractere**

```
FileWriter out = new FileWriter("fisier.txt");
```

- **crearea unui flux de intrare pe octeti**

```
FileInputStream in = new FileInputStream("fisier.dat");
```

- **crearea unui flux de iesire pe octeti**

```
FileOutputStream out = new  
    FileOutputStream("fisier.dat");
```

Fluxuri de procesare (1)

- responsabile cu preluarea datelor de la un flux primitiv și procesarea acestora pentru a le oferi într-o altă formă, mai utilă dintr-un anumit punct de vedere.
- "Bufferizare"
 - BufferedReader, BufferedWriter
 - BufferedInputStream, BufferedOutputStream
- Conversie octeți-caractere/caractere-octeți
 - InputStreamReader
 - OutputStreamWriter
- Concatenare
 - SequenceInputStream
- Serializare
 - ObjectInputStream, ObjectOutputStream

Fluxuri de procesare (2)

- Conversie tipuri de date de tip primitiv într-un format binar, independent de mașina pe care se lucrează
 - `DataInputStream`, `DataOutputStream`
- Numărare
 - `LineNumberReader`,
`LineNumberInputStream`
- Citire în avans
 - `PushbackReader`, `PushbackInputStream`
- Afișare
 - `PrintWriter`, `PrintStream`

Crearea unui flux de procesare

```
FluxProcesare numeFlux = new  
    FluxProcesare(fluxPrimitiv);
```

- crearea unui flux de intrare printr-un buffer
`BufferedReader in = new BufferedReader(new
 FileReader("fisier.txt"));`

//echivalent cu:

```
FileReader fr = new FileReader("fisier.txt");  
BufferedReader in = new BufferedReader(fr);
```

- crearea unui flux de iesire printr-un buffer
`BufferedWriter out = new BufferedWriter(new
 FileWriter("fisier.txt"));`

//echivalent cu:

```
FileWriter fo = new FileWriter("fisier.txt");  
BufferedWriter out = new BufferedWriter(fo);
```

Fluxuri pentru lucrul cu fişiere

FileReader, FileWriter - caractere

FileInputStream, FileOutputStream - octeti

Listing 1: Copierea unui fisier

```
import java.io.*;
public class Copiere {
    public static void main(String[] args) {

        try {
            FileReader in = new FileReader("in.txt");
            FileWriter out = new FileWriter("out.txt");

            int c;
            while ((c = in.read()) != -1)
                out.write(c);

            in.close();
            out.close();

        } catch (IOException e) {
            System.err.println("Eroare la operatiile cu fisiere!");
            e.printStackTrace();
        }
    }
}
```

Citirea și scrierea cu buffer

- BufferedReader, BufferedWriter
- BufferedInputStream, BufferedOutputStream
- Introduc un buffer (zonă de memorie) în procesul de citire/scriere a informațiilor.
- `BufferedOutputStream out = new BufferedOutputStream(new FileOutputStream("out.dat"), 1024);`
//1024 este dimensiunea bufferului
- Scopul: eficiența
 - Scade numărul de accesări ale dispozitivului extern
 - Crește viteza de execuție

Metoda flush

- golește explicit bufferul

```
BufferedWriter out = new BufferedWriter(new  
    FileWriter("out.dat"), 1024);  
    //am creat un flux cu buffer de 1024 octeti  
for(int i=0; i<1000; i++) out.write(i);  
    //bufferul nu este plin, in fisier nu s-a scris nimic  
out.flush();  
    //bufferul este golit, datele se scriu in fisier
```


Metoda readLine

```
BufferedReader br = new  
    BufferedReader(new FileReader("in"));  
String linie;  
while ((linie = br.readLine()) != null) {  
    ...  
    //proceseaza linie  
}  
br.close();
```

DataInputStream și DataOutputStream

- un flux nu mai este văzut ca o însiruire de octeți, ci de date primitive.
- scrierea datelor se face în format binar, ceea ce înseamnă că un fișier în care au fost scrise informații folosind metode writeX nu va putea fi citit decât prin metode readX.
- transformarea unei valori în format binar - serializare.
- permit serializarea tipurilor primitive și a șirurilor de caractere.

DataStream	DataStream
readBoolean	writeBoolean
readByte	writeByte
readChar	writeChar
readDouble	writeDouble
readFloat	writeFloat
readInt	writeInt
readLong	writeLong
readShort	writeShort
readUTF	writeUTF

Intrări formatate: java.util.Scanner



```
Scanner s=new Scanner(System.in);
```

```
String nume = s.next();
```

```
int varsta = s.nextInt();
```

```
double salariu = s.nextDouble();
```

```
s.close();
```

leșiri formate

PrintStream și PrintWriter :

- print, println
- format, printf

```
System.out.printf("%s %8.2f %2d\n", nume,  
    salariu, varsta);
```

- Formatarea șirurilor de caractere se bazează pe clasa `java.util.Formatter`.

Fluxuri standard de intrare și ieșire

- `System.in` - `InputStream`
- `System.out` - `PrintStream`
- `System.err` - `PrintStream`

- Afișarea informațiilor pe ecran:

`System.out.print (argument);`

`System.out.println(argument);`

`System.out.printf (format, argumente...);`

`System.out.format (format, argumente...);`

- Afișarea erorilor:

```
catch(Exception e) {
```

```
System.err.println("Exceptie:" + e);
```

```
}
```

Citirea datelor de la tastatură (1)

- Clasa `BufferedReader`

```
BufferedReader stdin = new BufferedReader(  
    new InputStreamReader(System.in));
```

```
System.out.print("Introduceti o linie:");
```

```
String linie = stdin.readLine()
```

```
System.out.println(linie);
```

- Clasa `DataInputStream`

```
DataInputStream stdin = new DataInputStream(  
    System.in);
```

```
System.out.print("Introduceti o linie:");
```

```
String linie = stdin.readLine()
```

```
System.out.println(linie);
```

- Clasa `java.util.Scanner` (1.5)

```
Scanner s=new Scanner(System.in);
```

Citirea datelor de la tastatură (2)

- Redirectarea fluxurilor standard: stabilirea unei alte surse decât tastatura pentru citirea datelor, respectiv alte destinații decât ecranul pentru cele două fluxuri de ieșire.
- `setIn(InputStream)` - redirectare intrare
- `setOut(PrintStream)` - redirectare ieșire
- `setErr(PrintStream)` - redirectare erori

```
PrintStream fis = new PrintStream( new  
    FileOutputStream("rezultate.txt"));  
System.setOut(fis);
```

```
PrintStream fis = new PrintStream(new  
    FileOutputStream("erori.txt"));  
System.setErr(fis);
```

Analiza lexicală pe fluxuri: StreamTokenizer

Listing 2: Citirea unor atomi lexicali dintr-un fisier

```
/* Citirea unei secvente de numere si siruri  
   dintr-un fisier specificat  
   si afisarea tipului si valorii lor  
   */  
  
import java.io.*;  
public class CitireAtomi {  
    public static void main(String args[]) throws IOException{  
  
        BufferedReader br = new BufferedReader(new FileReader("fisier.txt"));  
        StreamTokenizer st = new StreamTokenizer(br);  
  
        int tip = st.nextToken();  
        //Se citește primul atom lexical  
  
        while (tip != StreamTokenizer.TT_EOF) {  
            switch (tip) {  
                case StreamTokenizer.TT_WORD:  
                    System.out.println("Cuvant: " + st.sval);  
                    break;  
                case StreamTokenizer.TT_NUMBER:  
                    System.out.println("Numar: " + st.nval);  
            }  
  
            tip = st.nextToken();  
            //Trecem la următorul atom  
        }  
    }  
}
```


Clasa RandomAccessFile

- permite accesul nesecvențial (direct) la conținutul unui fișier;
- este o clasă de sine stătătoare, subclasă directă a clasei Object;
- se găsește în pachetul java.io;
- oferă metode de tipul readX, writeX;
- permite atât citirea cât și scriere din/în fișiere cu acces direct;
- permite specificarea modului de acces al unui fișier (read-only, read-write).

```
RandomAccessFile f1 = new  
    RandomAccessFile("fisier.txt", "r");  
//deschide un fisier pentru citire
```

```
RandomAccessFile f2 =new  
    RandomAccessFile("fisier.txt", "rw");  
//deschide un fisier pentru scriere si citire
```

Clasa RandomAccessFile (2)

- Program pentru afișarea pe ecran a liniilor dintr-un fișier text, fiecare linie precedată de numărul liniei și de un spațiu.

```
import java.io.*;
class A{
    public static void main(String arg[]) throws
        IOException{
        RandomAccessFile raf=new RandomAccessFile(
            arg[0],"r");

        String s;
        int i=1;
        while( (s=raf.readLine())!=null) {
            System.out.println(i+" "+s);
            i++;
        }
        raf.close();
    }
}
```

Clasa File (1)

- Clasa File nu se referă doar la un fișier ci poate reprezenta fie un fișier anume, fie mulțimea fișierelor dintr-un director.
- Utilitatea clasei File constă în furnizarea unei modalități de a abstractiza dependențele căilor și numelor fișierelor față de mașina gazdă
- Oferă metode pentru testarea existenței, ștergerea, redenumirea unui fișier sau director, crearea unui director, listarea fișierelor dintr-un director, etc.
- Constructorii fluxurilor pentru fișiere acceptă ca argument obiecte de tip File:
File f = new File("fisier.txt");
FileInputStream in = new FileInputStream(f);
- `String[] list()`
- `File[] listFiles()`
- `String getAbsolutePath()`

Clasa File (2)

- Listare fişiere dintr-un director dat, cu indentare, recursiv, în subdirectoare

```
import java.io.*;
import java.util.*;
class Dir{

public static void main (String arg[]) throws
    IOException {
    String dname = ".";
    if (arg.length ==1)  dname=arg[0];
    Dir d= new Dir();
    d.dirlist(new File(dname)," ");
}
}
```

Clasa File (3)

```
public void dirlist (File d, String sp) throws IOException
{
    String [] files=d.list(); //lista numelor din obiectul d
    if (files ==null ) return;
    String path =d.getAbsolutePath();
        //calea completa spre obiectul d
    for(int i=0;i<files.length;i++){
        File f = new File(d+"\\ "+files[i]);
        if (f.isDirectory()) {
            System.out.println (sp+path+"\\ "+files[i]);
            dirlist (f,sp+" ");
        }
        else System.out.println (sp+path+"\\ "+ files[i]);
    }
}
}
```

Agregare / Moștenire **Upcasting / Downcasting**

Programare Orientată pe Obiecte



Agregare și Compunere

- Se referă la prezența unei referințe către un obiect dintr-o clasă într-o altă clasă.
- **Agregarea** (aggregation) – obiectul container poate exista și în absența obiectelor agregate (care pot fi *null*) - *weak association*.
- **Compunerea** (composition) - este o agregare *strong* - existența unui obiect este dependentă de un alt obiect.
- La dispariția obiectelor conținute, existența obiectului container încetează.

Agregare și Compunere



Inițializarea obiectelor conținute poate fi făcută în 3 momente de timp distincte:

- la **definirea** obiectului (înaintea constructorului: folosind fie o valoare inițială, fie blocuri de inițializare)
- în cadrul **constructorului**
- chiar **înainte de folosire** (*lazy initialization*)

Exemplu

```
class SetAsVector_A{
    private Vector v=new Vector();
    // sau prin constructor:
    /*public SetAsVector_A(){
        v=new Vector(); }*/
    public boolean add(Object o){
        if(v.contains(o)) return false;
        return v.add(o);
    }
    public String toString(){
        return v.toString();
    }
    /* trebuiesc redefinite toate metodele pe
    care dorim sa le puna la dispozitie clasa!! */
}
```

Exemplu

```
public static void main(String arg[]){
    SetAsVector_A s1= new SetAsVector_A();
    s1.insertElementAt("abc",0);
    // trebuie definit insertElement At!!!!
    s1.insertElementAt("abc",0);
    System.out.println(s1);
    // trebuie definit toString!!!!
}
```

Moștenire (Inheritance)

- Numită și **derivare**
- mecanism de refolosire a codului specific limbajelor orientate obiect
- 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**.

Exemplu

```
class SetAsVector_M extends Vector{  
    public boolean add(Object o){  
        if(contains(o)) return false;  
        return super.add(o);  
    }  
}
```

**/* trebuie redefinite toate metodele din clasa
Vector care adauga sau modifica o valoare!!!*/**

```
}
```

....

```
public static void main(String arg[]){  
    SetAsVector_M s1= new SetAsVector_M();  
    s1.insertElementAt("abc",0);  
    s1.insertElementAt("abc",0);  
    System.out.println(s1); }  
}
```

Agregare vs. moștenire

Când se folosește moștenirea și când compunerea?

- Depinde de datele problemei analizate dar și de concepția designerului,
- **Agregarea** - folosită atunci când se dorește folosirea trăsăturilor unei clase în interiorul altei clase, dar nu și interfața sa (prin moștenire, noua clasă ar expune și metodele clasei de bază).

Ex: Implementarea funcționalității obiectului conținut în noua clasă și **limitarea** acțiunilor utilizatorului la metodele din noua clasă (să nu se permită utilizatorului folosirea metodelor din vechea clasă) - SetAsVector

- noua clasă va conține un obiect de tipul clasei conținute, cu specificatorul de acces private.

Agregare vs. moștenire

Când se folosește moștenirea și când compunerea?

- **Moștenirea** permite crearea unor versiuni “specializate” ale unor clase existente (de bază).
- Moștenirea 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.

Exemplu:

clasa Patrat care moștenește clasa Patrulater.

Diferența dintre moștenire și agregare

- Este de fapt diferența dintre cele 2 tipuri de relații majore prezente între obiectele unei aplicații:
- **is a** - indică faptul că o clasă este derivată dintr-o clasă de bază

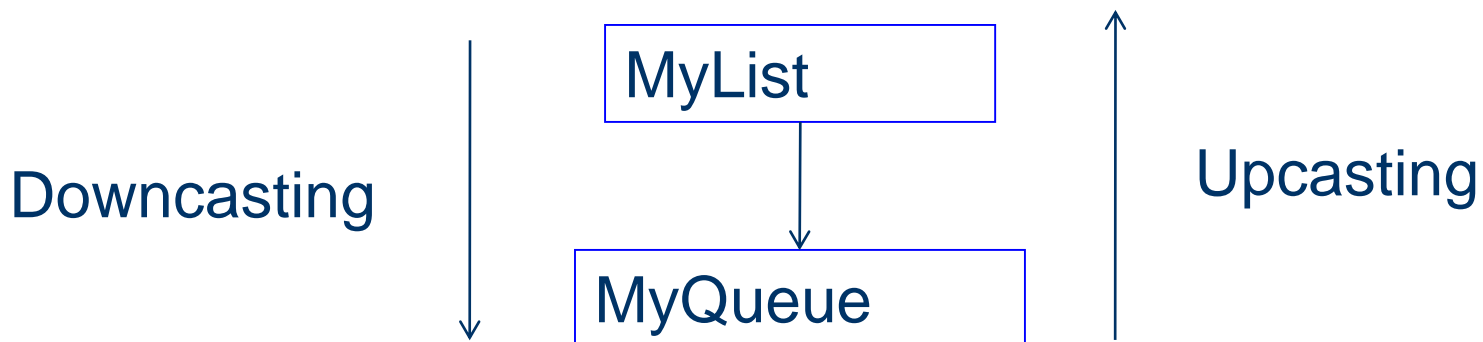
Ex: Având o clasă Animal și o clasă AnimalDomestic, atunci AnimalDomestic va fi derivat din Animal, cu alte cuvinte AnimalDomestic **is an** Animal

- **has a** - indică faptul că o clasă-container are o clasă conținută în ea

Ex: Dacă avem o clasă Masina și o clasă Motor, atunci Motor va fi referit în cadrul Masina, cu alte cuvinte Masina **has a** Motor

Upcasting - Downcasting

```
class MyList { }  
class MyQueue extends MyList { }  
....  
MyList a ;  
MyQueue q= new MyQueue();  
a = q;                // upcasting  
q = (MyQueue) a ;     // downcasting
```



Upcasting

- **Convertirea** unei referințe la o clasă derivată într-una a unei clase de bază
- Realizat **automat** (nu trebuie declarat explicit de către programator)
- Exemplu de upcasting:

```
class Patrati extends Poligon { ... }
```

```
Poligon metoda1( ) {  
    Poligon p = new Poligon();  
    Patrati t = new Patrati();  
    if (...) return p; // Corect  
    return t; // upcast automat!!! – corect!  
}
```

Exemplu

```
class A { }
class B extends A { }
public class Test {
    public void method(A a) {
        System.out.println("Method A"); }
    public void method(B b) {
        System.out.println("Method B"); }
    public static void main(String[] args) {
        Test t=new Test();
        B b = new B();
        t.method(b); // "Method B"
        // upcasting a B into an A:
        t.method((A) b); // "Method A"
    }
}
```

Downcasting

- operația **inversă** upcast-ului
- conversie explicită de tip în care se merge în **jos** pe ierarhia claselor (se convertește o clasă de bază într-una derivată)
- trebuie făcut **explicit** de către programator.

```
boolean equals(Object o) {  
    If (!(o instanceof Car)) return false;  
    Car other = (Car)o;  
    // compare this to other and return  
}
```
- **posibil** numai dacă obiectul convertit este, de fapt, de tipul clasei către care se face downcasting-ul. Altfel, mașina virtuală aruncă o excepție la rularea programului.

Exemplu

```
public class Person {  
    private String name;  
    private int age;  
    public boolean equals(Object anObject) {  
        if (anObject == null) return false;  
        /* testare daca este de acelasi tip */  
        if (getClass( ) != anObject.getClass())  
            return false;  
        /* downcast - Object  clasa parinte a  
            oricarei clase */  
        Person aPerson = (Person) anObject;  
        return name.equals(aPerson.name) &&  
            (age == aPerson.age);  
    }  
}
```

Exemplu

```
public String printAll(LinkedList c) {  
    Object arr[]=c.toArray();  
    String list_string="";  
    for(int i=0;i<c.size();i++) {  
        String mn=(String)arr[i];  
        list_string+=(mn);  
    }  
    return list_string;  
}
```

Quizz

```
class Patrat extends Poligon { ... }
```

Varianta 1:

```
Patrat metoda2( ) {  
    Poligon p = new Poligon();  
    Patrat t = new Patrat();  
    if (...)  
        return p; // Eroare!!  
    else  
        return t; // Corect  
}
```

Varianta 2:

```
Patrat metoda2( ) {  
    Poligon p = new Poligon();  
    Patrat t = new Patrat();  
    if (...)  
        return (Patrat)p; // corect? Se execută?  
    else  
        return t; // Corect  
}
```