Fluxuri

Programare Orientată pe Obiecte

Fluxuri

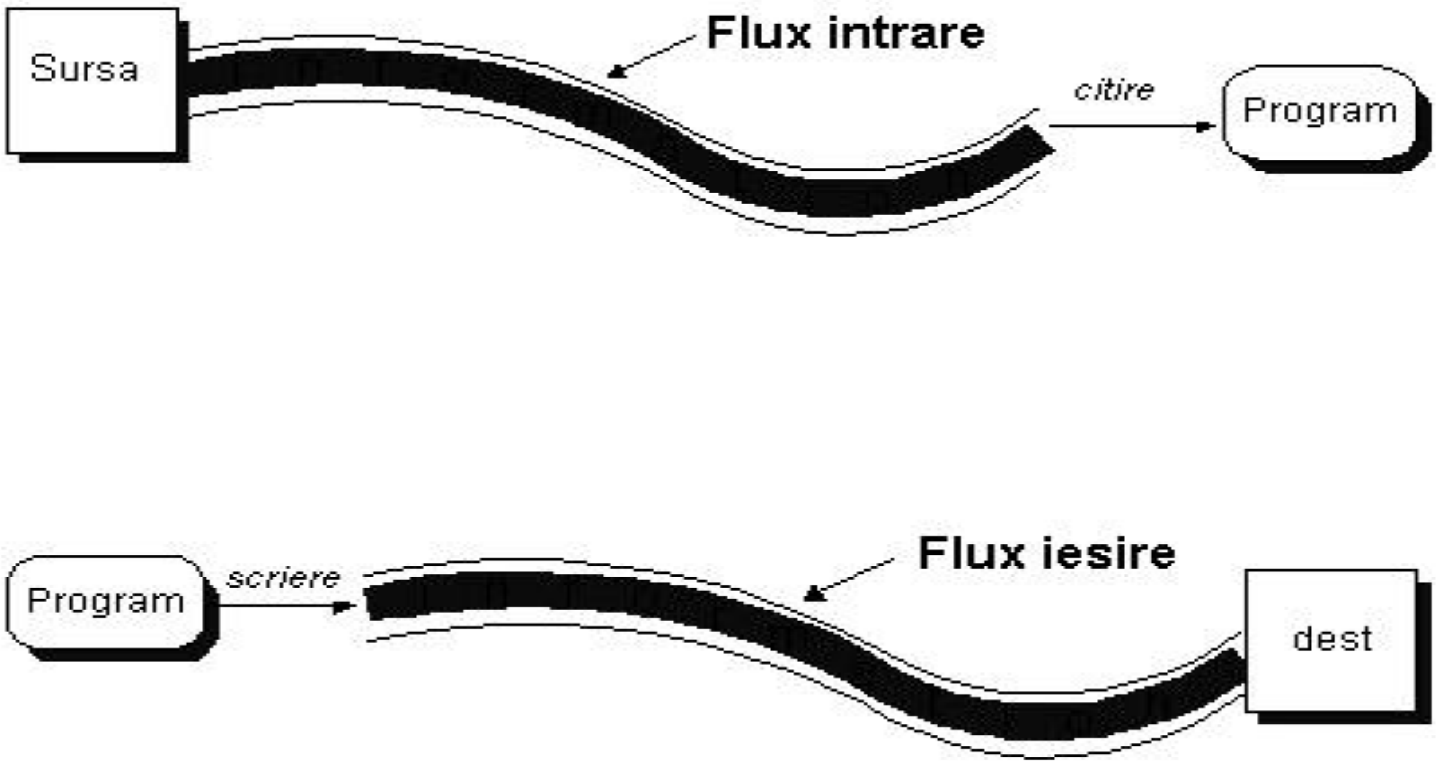
- Ce sunt fluxurile ?
- Clasificare, ierarhie
- Fluxuri primitive
- Fluxuri de procesare
- Intrări şi ieşiri formatate
- Fluxuri standard de intrare şi ieşire
- Analiza lexicală
- Clase independente
  (RandomAccessFile, File)

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 citeşte date se numeşte flux de intrare.
Un flux care scrie date se numeşte flux de ieşire.

Ce sunt fluxurile? (3)

- Pachetul care oferă suport pentru operațiile de intrare/ieşire: 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 ieşire (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:

- Fişier
  - FileReader, FileWriter,
  - FileInputStream, FileOutputStream
- Memorie
  - CharArrayReader, CharArrayWriter,
  - ByteArrayInputStream,
  - ByteArrayOutputStream,
  - StringReader, StringWriter
- Pipe
  - PipedReader, PipedWriter,
  - PipedInputStream, PipedOutputStream
  - folosite pentru a canaliza ieşirea 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 şi 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 maşina pe care se lucrează
  - DataInputStream, DataOutputStream
- Numărare
  - LineNumberReader, LineNumberInputStream
- Citire în avans
  - PushbackReader, PushbackInputStream
- Afişare
  - 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 şi 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
  - Creşte viteza de execuție

Metoda flush

- goleşte 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 şi 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 fişier 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 şi a şirurilor de caractere.

DataInputStream	DataOutputStream
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();
```

Ieşiri formatate

PrintStream şi PrintWriter :

- print, println
- format, printf

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

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

Fluxuri standard de intrare şi ieşire

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

- Afişarea informațiilor pe ecran:

```
System.out.print (argument);
System.out.println(argument);
System.out.printf (format, argumente...);
System.out.format (format, argumente...);
```

- Afişarea 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 ieşire.

- setIn(InputStream) - redirectare intrare
- setOut(PrintStream) - redirectare ieşire
- 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 citeste 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 urmatorul atom
    }
  }
}
```

Clasa RandomAccessFile

- permite accesul nesecvențial (direct) la conținutul unui fişier;
- este o clasă de sine stătătoare, subclasă directă a clasei Object;
- se găseşte în pachetul java.io;
- oferă metode de tipul readX, writeX;
- permite atât citirea cât şi scriere din/în fişiere cu acces direct;
- permite specificarea modului de acces al unui fişier (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 afişarea pe ecran a liniilor dintr-un fişier text, fiecare linie precedată de numărul liniei şi 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 fişier ci poate reprezenta fie un fişier anume, fie mulțimea fişierelor dintr-un director.**
- **Utilitatea clasei File constă în furnizarea unei modalități de a abstractiza dependențele căilor şi numelor fişierelor față de maşina gazdă**
- **Oferă metode pentru testarea existenței, ştergerea, redenumirea unui fişier sau director, crearea unui director, listarea fişierelor dintr-un director, etc.**
- **Constructorii fluxurilor pentru fişiere 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 / Moştenire
Upcasting / Downcasting

Programare Orientată pe Obiecte

Agregare şi Compunere

- Se referă la prezența unei referințe către un obiect dintr-o clasă într-o altă clasă.

- **Agregarea** (aggregation) – obiectul container poate exista şi 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 şi 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!!!!
}
```

Moştenire (Inheritance)

- Numită şi **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 şi 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 numeşte **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. moştenire

Când se foloseşte moştenirea şi când compunerea?

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

Ex: Implementarea funcționalității obiectului conținut în noua clasă şi **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. moştenire
Când se foloseşte moştenirea şi când compunerea?

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

- Moştenirea folosită în general atunci când se doreşte 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 moşteneşte clasa Patrulater.

Diferența dintre moştenire şi 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

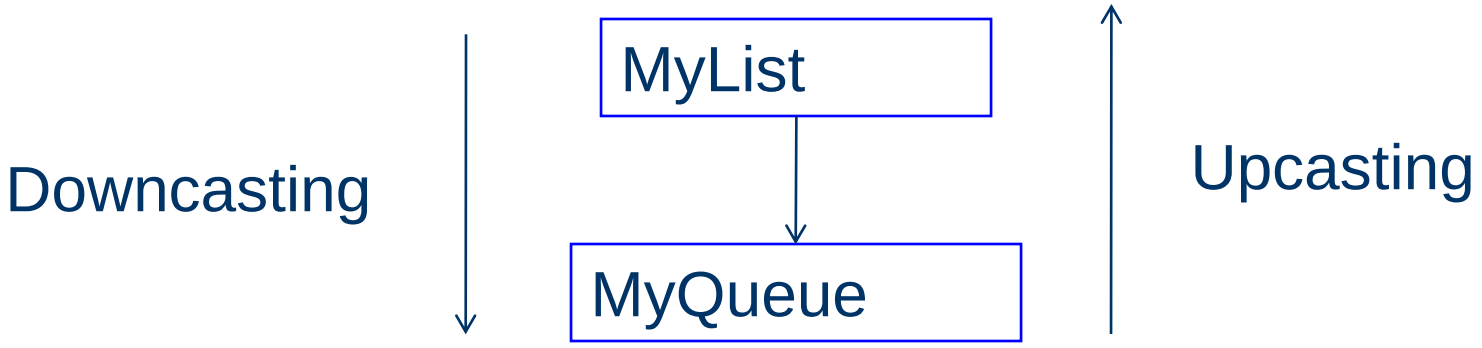
Ex: Având o clasă Animal şi o clasă AnimalDomestic, atunci AnimalDomestic va fi derivat din Animal, cu alte cuvinte AnimalDomestic *is an* Animal

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

Ex: Dacă avem o clasă Masina şi 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 Patrat extends Poligon { … }

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

Exemplu

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

Downcasting

- operația **inversă** upcast-ului
- conversie explicită de tip în care se merge în **jos** pe ierarhia claselor (se converteşte 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 declarat ca fiind de o clasă de bază este, de fapt, **instanță** a clasei derivate către care se face downcasting-ul. Altfel, maşina 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
}
```

Legare statică/dinamică – static/dynamic binding în Java

Programare Orientată pe Obiecte

Polimorfism

- **Polimorfism – abilitatea unui obiect de a se comporta diferit la acelaşi mesaj**
- **Două tipuri: static şi dinamic**

- **Supraîncărcarea (overloading)**
- **Supradefinirea (overriding)**

```
class A {
    void metoda() {
        System.out.println("A: metoda fara parametru");
    }
    // Supraîncărcare
    void metoda(int arg) {
        System.out.println("A: metoda cu un parametru");
    }
}
class B extends A {
    // Supradefinire
    void metoda() {
        System.out.println("B: metoda fara parametru");
    }
}
```

Static/dynamic binding în Java

- Binding – procesul de a stabili ce metodă sau variabilă va fi apelată

- **Static binding şi dynamic binding - două concepte importante în Java**
- Legate direct de execuția codului
- Mai multe metode cu acelaşi nume (method overriding) sau două variabile cu acelaşi nume în aceeaşi ierarhie de clase, care este utilizată?
- Majoritatea referințelor sunt rezolvate în timpul compilării, dar cele care depind de obiect şi polimorfism sunt rezolvate la execuție, atunci când este de fapt disponibil obiectul.

- Dynamic binding – late binding (la execuție)
- Static binding – early binding (la compilare)

Static/dynamic binding în Java

- *Diferența între static şi dynamic binding* în Java
- Întrebare populară la angajarea în domeniu
- Explorează cunoştințele candidaților legate de cine determină ce metodă va fi apelată dacă există mai multe metode cu acelaşi nume, ca în cazul metodelor supraîncărcate sau supradefinite (method overloading and overriding).
- Compilatorul sau JVM – maşina virtuală Java?

Legare statică/dinamică – Static/dynamic binding în Java

```
class Vehicle {
    public void drive() {
        System.out.println("A");
    }
}

class Car extends Vehicle {
    public void drive() {
        System.out.println("B");
    }
}
class TestCar {
    public static void main(String args[]) {
        Vehicle v;
        Car c;
        v = new Vehicle();
        c = new Car();
        v.drive();
        c.drive();
        v = c;
        v.drive();
    }
}
```

Polimorfism

- Legarea dinamică are loc în Java pentru orice metodă care nu are atributul *final* sau *static*, metodă numită polimorfică.
- In Java majoritatea metodelor sunt polimorfice.
- Metodele polimorfice Java corespund functiilor virtuale din C++.
- Apelul functiilor polimorfice este mai putin eficient ca apelul functiilor cu o singura formă.
- Fiecare clasă are asociat un tabel de pointeri la metodele (virtuale) ale clasei.

Static Binding vs Dynamic binding în Java

Diferențe între legarea statică şi cea dinamică în Java:

- **Static binding** se realizează la **Compilare** în timp ce **Dynamic binding** se realizează la **Rulare** - Execuție.

- Metodele şi variabilele **private**, **final** sau **static** utilizează static binding în timp ce metodele **virtuale - abstracte** utilizează dynamic binding.

- **Static binding** se face pe baza informației legate de **Tip** (**clasa** în Java), în timp ce **Dynamic binding** utilizează **Obiectul** pentru a realiza legarea.

- **Static binding** este folosit frecvent la metodele supraîncărcate (**overloaded** methods), **Dynamic binding** (dynamic dispatch) este asociat în general cu metodele supradefinite (**overriding** methods).

Static Binding: Exemplu Java

```
public class StaticBindingTest {
    public static void main(String args[])  {
          Collection c = new HashSet();
          StaticBindingTest et = new StaticBindingTest();
          et.sort(c);
     }
  // metoda supraincarcata cu argument Collection
    public Collection sort(Collection c){
          System.out.println("In metoda sort(Collection)!");
          return c;
       }
   /*metoda supraincarcata cu argument HashSet, subclasa
    a lui Collection */
   public Collection sort(HashSet hs){
          System.out.println("In metoda sort(HashSet )!");
          return hs;
       }
    }
```

Output: In metoda sort(Collection)!

Dynamic Binding: Exemplu Java

```
class Vehicle {
    public void start() {
            System.out.println("In metoda start din Vehicle!");
    }
}
class Car extends Vehicle {
    public void start() {
            System.out.println ("In metoda start din Car!");
    }
}
public class DynamicBindingTest {
  public static void main(String args[]) {
     Vehicle vehicle = new Car(); //tip Vehicle, dar obiect Car
     vehicle.start();   //start din clasa Car - start() e supradef
    }
}
```

Output: In metoda start din Car!

Dynamic Binding

- Conceptul de overriding
- Car extends Vehicle - supradefineşte start()
- Apelul lui start() pentru un obiect Vehicle, apelează start() din subclasa Car deoarece obiectul referit de tipul Vehicle este un obiect Car
- Aceasta se petrece la execuție pentru că obiectul este creat doar la execuție -> Dynamic binding in Java.
- Dynamic binding este mai încet decât static binding pentru că apare în momentul execuției şi necesită timp pentru a determina care metodă este apelată de fapt.

Exemplu Static Binding vs Dynamic binding

```
public class Animal {
        public String type = "mamifer";
        public void show() {
            System.out.println(“Animalul este un: " + type);
        }
}
public class Dog extends Animal {
        public String type;
        public Dog(String type){
            this.type = type;
        }
        public void show() {
            System.out.println(“Cainele este un: " + type);
        }
}
```

```
…
Animal doggie = new Dog(“ciobanesc");
 doggie.show();
System.out.println (“Tipul este: " + doggie.type);
....
```

Exemplu Static Binding vs Dynamic binding

Output:

“Cainele este un: ciobanesc“ (dynamic binding)

“Tipul este: mamifer" (static binding)

Observații

- Datele nu sunt niciodată supradefinite, doggie.type foloseşte Animal.type -- "static binding“

Avantaj: comportamentul este legat de tipul obiectului invocat, fără ca noi să ştim precis tipul acestuia.

Exemplu: Daca trimitem un Animal, nu ştim dacă este Cat/Dog/altceva, dar el va adopta comportamentul adecvat:

```
public void makeNoise(List<Animal> animals) {
      for (Animal a : animals) {
         a.makeNoise();
      }
}
```

- Fiecare animal din lista va produce propriul zgomot (va mieuna, lătra, etc)
- **Observație**: Clasa Animal poate fi abstracta. Astfel, comportamentul va fi definit de clasele concrete derivate din ea.

Exemplu

```
public class Shape {
    int x= 10;
    void draw() {
        System.out.println(“Shape”);
    }
}
public class Circle extends Shape{
    int x=5;
    void draw(){
        System.out.println(“Circle”);
    }
}
public class Test{
    public static void main(String[] args) {
        Shape shape = new Circle();
        shape.draw(); // DYNAMIC BINDING
        System.out.println(shape.x); // STATIC BINDING
    }
}
```

Explicații

- Atunci când compilatorul vede apelul draw(), el ştie că obiectul respectiv este de tipul Shape, dar el ştie şi că acel obiect poate fi o referință la orice clasă derivată din Shape.
- De aceea, compilatorul nu ştie ce versiune a metodei draw() este de fapt apelată. Aceasta se va şti doar în momentul execuției instrucțiunii respective:
- **shape.draw();** - metoda draw din Circle

- În unele cazuri, compilatorul poate determina versiunea corectă.
- În Java, variabilele membru prezintă static binding, deoarece Java nu permite comportament polimorfic pentru variabilele membru.
- Aceasta înseamnă că atât clasa Shape cât şi clasa Circle au o câte o variabilă membru cu acelaşi nume:

System.out.println(shape.x); - valoarea lui x din Shape.

Concluzie - dynamic binding

- metodele suprascrise în clasele derivate vor fi apelate folosind dynamic binding (late binding).
- este un mecanism prin care compilatorul, în momentul în care nu poate determina implementarea unei metode în avans, lasă la latitudinea JVM-ului (maşinii virtuale) alegerea implementării potrivite, în funcție de tipul real al obiectului.
- această legare a implementării de numele metodei la **momentul execuției** stă la baza polimorfismului.

Ce va afişa următorul program?

```
class TestEgal{
    public boolean equals ( TestEgal other ) {
        System.out.println( "In equals din TestEgal" );   return false;
    }
    public static void main( String [] args ) {
        Object t1 = new TestEgal(), t2 = new TestEgal();
        TestEgal t3 = new TestEgal();
        Object o1 = new Object();
        int count = 0;
        System.out.println( count++ ); // afiseaza 0
        t1.equals( t2 ) ;
        System.out.println( count++ ); // afiseaza 1
        t1.equals( t3 );
        System.out.println( count++ ); // afiseaza 2
        t3.equals( o1 );
        System.out.println( count++ ); // afiseaza 3
        t3.equals(t3);
        System.out.println( count++ ); // afiseaza 4
        t3.equals(t2);
    }
}
```

Exercițiu propus

- Cum ar trebui să fie definite clasele Adult, Student şi Inginer astfel încât următoarea secvență să dea mesajele din comentarii, la **compilare**?

```
class Test {
   public static void main(String
   args[]) {
     Adult  a = new Student(); /* fara
                          eroare */
     Adult b = new Inginer(); /* fara
                          eroare */
     a.explorare(); // fara eroare
     b.explorare(); // fara eroare
     a.afisare();  //fara eroare
     b.afisare();  //eroare la compilare
   }
}
```