

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)

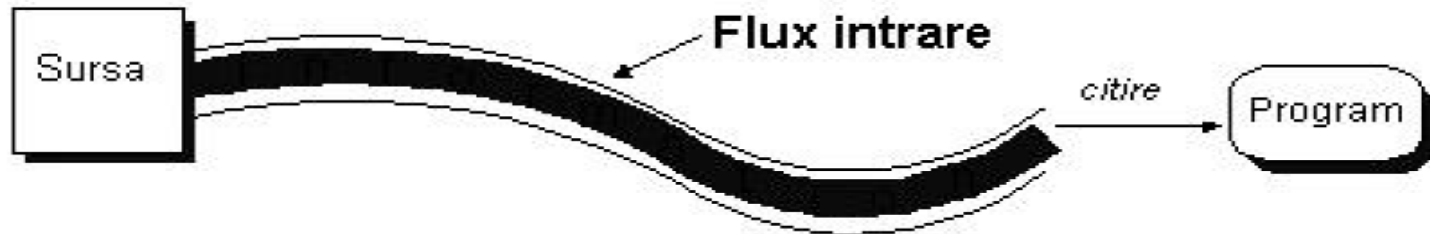
Ce sunt fluxurile? (1)



- 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.

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();
```

leșiri formateate

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 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 nesequențial (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]);
    }
}
}
```

Legare statică/dinamică – static/dynamic binding în Java

Programare Orientată pe Obiecte



Polimorfism

- Polimorfism – abilitatea unui obiect de a se comporta diferit la același mesaj
- Două tipuri: static și 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

- **Static binding și dynamic binding - două concepte importante în Java**
- Legate direct de execuția codului
- Mai multe metode cu același nume (method overriding) sau două variabile cu același nume în aceeași ierarhie de clase, care este utilizată?
- Binding – procesul de a stabili ce metodă sau variabilă va fi apelată
- Majoritatea referințelor sunt rezolvate în timpul compilării, dar cele care depind de Obiect și 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 și dynamic binding în Java*
- Întrebare populară la angajarea în domeniu
- Explorează cunoștințele candidaților legate de cine determină ce metodă va fi apelată dacă există mai multe metode cu același nume, ca în cazul metodelor supraîncărcate și supradefinite (method overloading and overriding).
- Compilatorul sau JVM – mașina 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();
    }
}
```

Static Binding vs Dynamic binding în Java

Diferențe între legarea statică și cea dinamică în Java:

- **Static binding** se realizează la **Compilare** în timp ce **Dynamic binding** se realizează la **Rulare - Execuție**.
- Metodele și 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 - supradefinește 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 și 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` folosește `Animal.type` -- "static binding"

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

Exemplu: Dacă trimitem un `Animal`, nu știm 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 abstractă. 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 știe că obiectul respectiv este de tipul `Shape`, dar el știe și că acel obiect poate fi o referință la orice clasă derivată din `Shape`.
- De aceea, compilatorul nu știe ce versiune a metodei `draw()` este de fapt apelată. Aceasta se va ști 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 și clasa `Circle` au o câte o variabilă membru cu același nume:
`System.out.println(shape.x)`; - valoarea lui `x` din `Shape`.

Ce va afișa 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 și Inginer astfel încât următoarea secvență să dea eroare la compilare doar unde este specificat?

```
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!
    }
}
```