

Metode avansate de programare

Curs3

- ☐ Şablonul de proiectare Singleton
- ☐ Şablonul de proiectare Decorator

- ☐ Excepții
- ☐ I/O

Șablonul de proiectare Singleton

- Permite restricționarea numărului de instanțieri ale unei clase la un singur obiect
- Utilizare:
 - în situații în care avem obiecte care trebuie accesate din mai multe locuri ale aplicației (în locul variabilor globale)
 - împreună cu alte pattern-uri: **Factory**, Builder, Prototype (ex. ne dorim un singur obiect factory pentru a crea obiecte de un anumit tip)
- *Exemple de utilizări ale șablonului Singleton:*
 - pentru obiecte de tip Factory (seminarul 1-2)
 - obiecte de tip logger
 - obiecte care reprezintă resurse partajate (conexiuni, sockets etc.)
 - obiecte ce conțin configurații pentru aplicație
- *Limitări:*
 - Încalcă principiul de proiectare Low-Coupling
 - In TDD este considerat *anti-pattern*
- *Recomandări:*
 - A nu se folosi in exces

Implementare

- În Java, pentru a asigura o singură instanțiere a clasei:
 - constructorul trebuie să fie *private*
 - instanța să fie oferită printr-o metodă statică, publică (`getInstance`)
- Există două abordări:
 - “Lazy object instantiation” – obiectul este încărcat în memorie doar dacă este folosit
 - “Eager object instantiation” – obiectul este încărcat în memorie în cazul în care poate va fi folosit
- **Assignment:** Creați clasa *Government*, singleton, folosind ambele abordări: lazy și eager. Discutați diferențele de implementare. Why: Lazy instantiation: dangerous in multithreaded applications

Singleton	
-	<code>singleton : Singleton</code>
-	<code>Singleton()</code>
+	<code>getInstance() : Singleton</code>

```
package singleton;
```

```
public class GovernmentLazy {  
    private GovernmentLazy() {}  
    static private GovernmentLazy instance = null;  
    public static GovernmentLazy instance()  
    {  
        if(instance==null) //Lazy instantiation  
            instance= new GovernmentLazy();  
        return instance;  
    }  
    //public methods
```

```
//    public static GovernmentLazy instance2()  
//    {  
//        if(instance==null) //don't want to block here  
//        {  
//            // cateva thread-uri pot sa astepte aici  
//            synchronized(GovernmentLazy.class)  
//            {  
//                // trebuie verificat din nou  
//                if(instance==null)  
//                    instance= new GovernmentLazy();// pare sigur  
//            }  
//        }  
//        return instance;  
//    }  
}
```

```
public class GovernmentEager  
{  
    private GovernmentEager() {}  
    static final private GovernmentEager instance = new  
        GovernmentEager();  
    static public GovernmentEager instance()  
    {  
        return instance;  
    }  
    //public methods  
}
```

Excepții

- ❑ O excepție este o situație “anormală” ce are loc în timpul execuției programului.

Tratarea excepțiilor nu mai este o opțiune ci o constrângere!!!

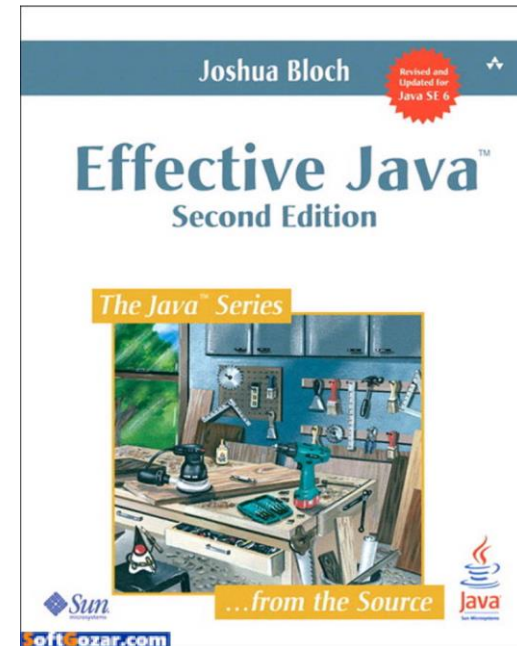
- ❑ Folosite corect și eficient, excepțiile îmbunătățesc attributele de calitate ale sistemelor soft(readability, reliability, maintainability, . . .)

Item 57: Use exceptions only for exceptional conditions

```
// Horrible abuse of exceptions. Don't ever do this!
```

```
try {  
    int i = 0;  
    while(true)  
        range[i++].climb();  
} catch(ArrayIndexOutOfBoundsException e) {  
}
```

```
for (Mountain m : range)  
    m.climb();
```

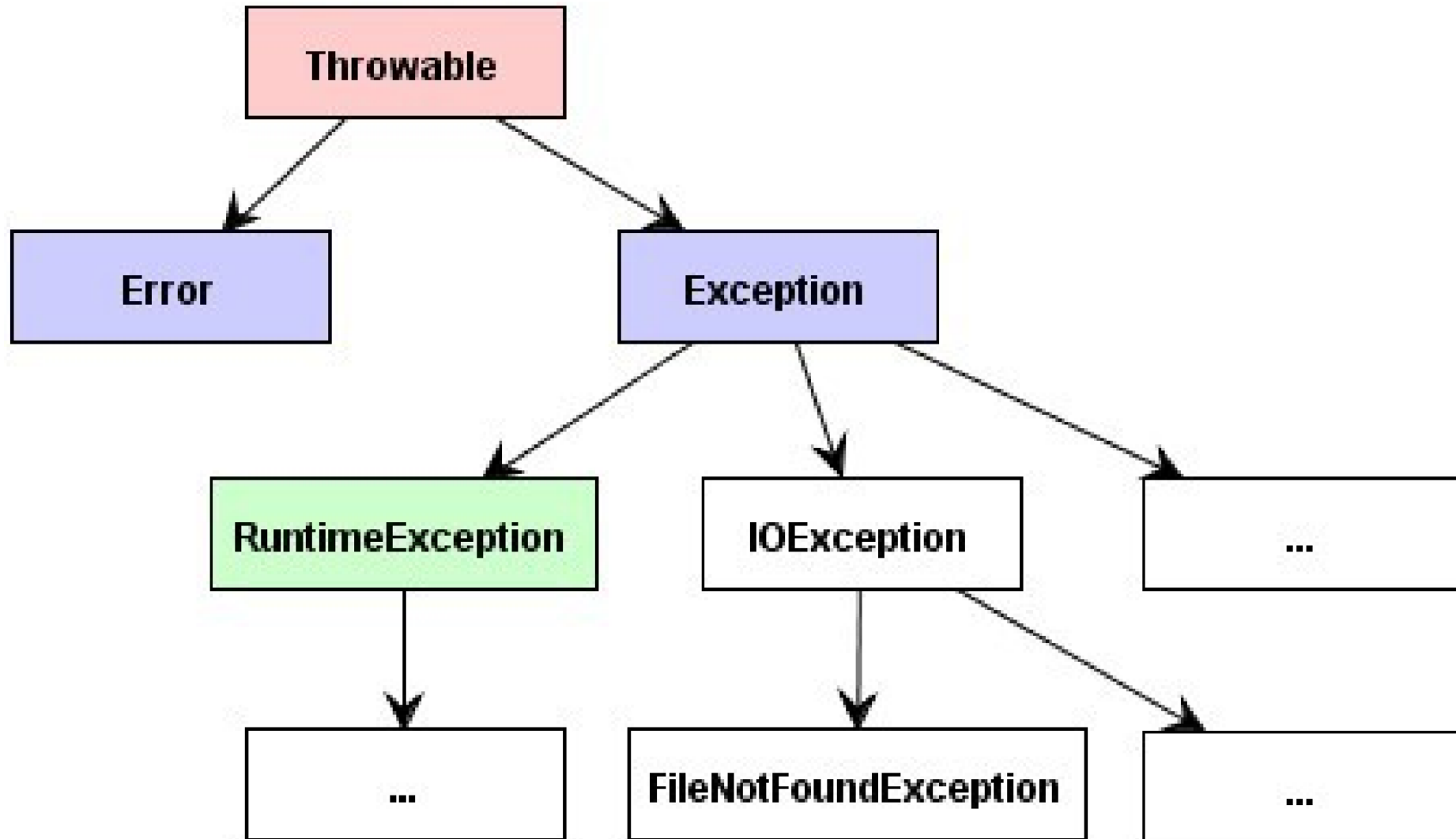


Exemplu

```
public static void main(String[] args)
{
    int i=Integer.parseInt("12b");
}
```

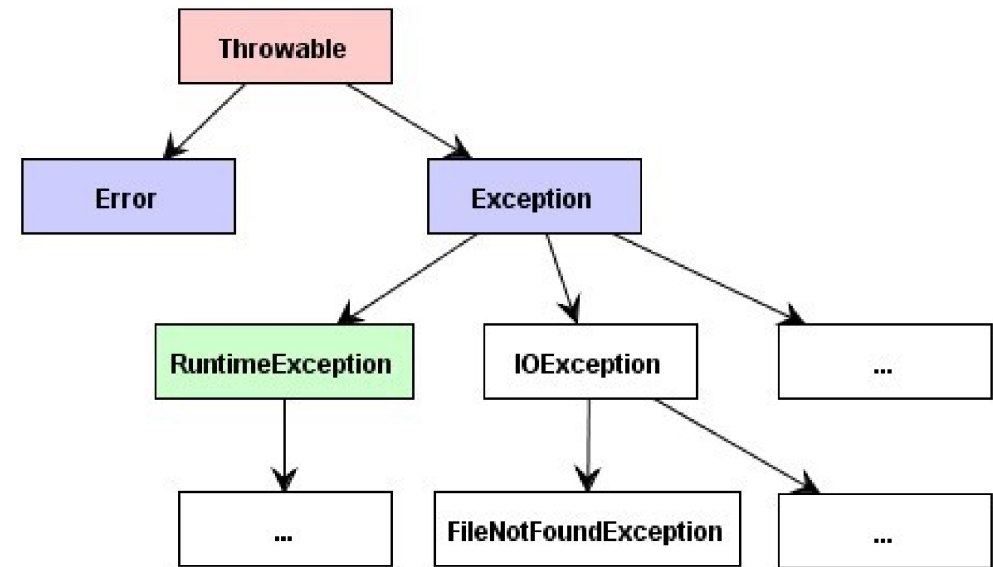
```
Exception in thread "main" java.lang.NumberFormatException: For input string: "12b"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.lang.Integer.parseInt(Integer.java:580)
    at java.lang.Integer.parseInt(Integer.java:615)
    at exceptions.Ex1.main(Ex1.java:10) <5 internal calls>
```

Ierarhia claselor ce definesc excepții



Tipuri de excepții

- Clasele **Error** și **RuntimeException**, împreună cu descendenții lor, formează categoria excepțiilor **neverificate (unchecked)**, adică excepții care pot fi generate, fără obligativitatea ca ele să apară în clauze **throws**.



- Restul excepțiilor sunt **verificate (checked)**, adică **la compilare se verifică dacă există clauze throws corespunzătoare**.

Instructiunea throw – emiterea unei excepții

- O instrucțiune **throw** poate să apară într-o funcție numai dacă:
 - ea se găsește în interiorul unui bloc **try-catch** care captează (try-catch) tipul de excepție generată de expresia din throwsau
 - definiția funcției este însoțită de o clauza throws în care apare tipul de excepție respectiv (**excepții verificate**)sau
 - excepția generată aparține claselor **RuntimeException** sau **Error**, sau descendenților acestora

Tratarea excepțiilor try - catch - finally

```
try {  
    // Bloc de instructiuni  
    metodaX()  
    metodaY()  
    metodaZ()  
}  
catch (TipExceptie1 variabila) {  
    // Tratarea exceptiilor de tipul 1  
}  
catch (TipExceptie2 variabila) {  
    // Tratarea exceptiilor de tipul 2  
}  
catch (TipExceptie3 | TipExceptie4 variabila) {  
    // Tratarea exceptiilor de tipul 3 sau 4  
}  
finally {  
    // Cod care se executa indiferent daca apar sau nu exceptii  
}  
...execuția continuă
```

→ S-a generat o excepție

```
// Java 7  
try {  
    ...  
} catch (IOException | FileNotFoundException ex) {  
    ...  
}
```

Avantajele folosirii excepțiilor

1. Separarea codului pentru tratarea unei erori de codul în care ea poate să apară;
2. Propagarea unei excepții până la un analizor de excepții corespunzător;
3. Gruparea erorilor după tipul lor.

1. Separarea codului

```
int openResult = open();

if (openResult == FILE_NOT_FOUND) {
    // handle error
} else if (openResult == INSUFFICIENT_PERMISSIONS) {
    // handle error
} else { // SUCCESS
    int readResult = read();
    if (readResult == DISK_ERROR) {
        // handle error
    } else {
        // SUCCESS
    }
    ...
}
```

```
try {
    open();
    read();
    ...
} catch (FILE_NOT_FOUND) {
    // handle error
} catch (INSUFFICIENT_PERMISSIONS) {
    // handle error
} catch (DISK_ERROR) {
    // handle error
}
```

2. Propagarea excepțiilor

```
int metoda3() throws TipExcepție {
    ...
    throw new TipExcepție();
    ...
}
int metoda2() throws TipExcepție {
    ...
    metoda3();
    ...
}
int metoda1() {
    try {
        metoda2();
    } catch (TipExcepție e) {
        //proceseaza excepție
    }
}
```

- O metoda poate să nu își asume responsabilitatea tratării excepțiilor apărute în cadrul ei!

3. Gruparea erorilor după tipul lor

```
try {  
    String driverName = new String(Files.readAllBytes(Paths.get("driver.txt")));  
    Class.forName(driverName).newInstance();  
} catch (IOException ex) {  
    // probleme cu fisierul din care vrem sa citim  
} catch (ClassNotFoundException ex) {  
    // nu exista clasa driver  
} catch (IllegalAccessException ex) {  
    // lipsa acces clasa  
} catch (InstantiationException ex) {  
    // clasa nu poate fi instantiata  
}
```

Definirea propriilor clase de excepții

- Extinderea unei clase existente din ierarhia de clase ce are ca rădăcina Throwable
- Decizie: Checked vs. Unchecked

```
public class ExceptieProprie extends RuntimeException {  
    //Proprietati si constructori  
    public ExceptieProprie(String mesaj) {  
        super(mesaj); // Apeleaza constructorul superclasei }  
    }  
}
```

Definirea propriilor clase de excepții

Dezavantajle derivării excepțiilor proprii din clasa Exception

```
public class ValidatorException extends Exception {  
    public ValidatorException(String message) { super(message); }  
}
```

```
public class StudentValidator implements Validator<Student> {  
    @Override  
    public void validate(Student e) throws ValidatorException {  
        String errMsg="";  
        if (e.getId() == null || "".equals(e.getId()))  
            errMsg+="Id error ";  
        if (e.getFirstName() == null || "".equals(e.getFirstName()))  
            errMsg+="first name error ";  
        if (e.getLastName() == null || "".equals(e.getLastName()))  
            errMsg+="last name error ";  
        if (e.getEmail() == null || "".equals(e.getEmail()))  
            errMsg+="email error error ";  
        if (errMsg!="")  
            throw new ValidatorException(errMsg);  
    }  
}
```


Excepțiile în contextul moștenirii

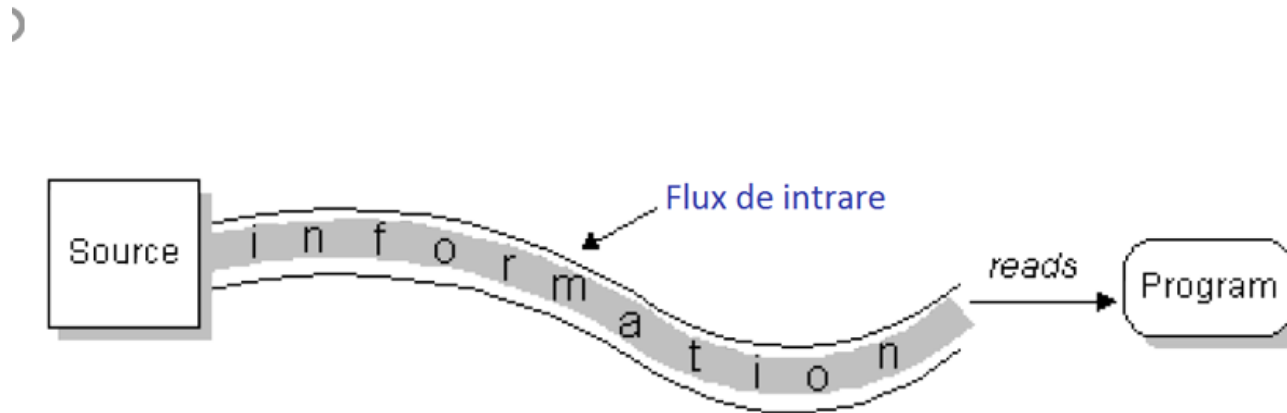
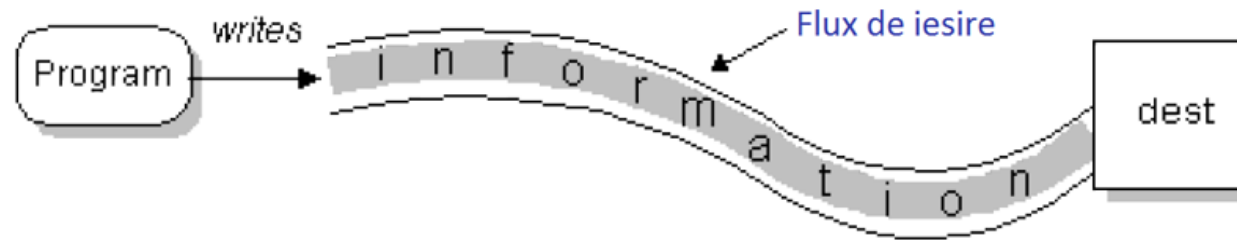
- Metodele suprascrise (overriden) pot arunca **numai** excepțiile specificate de metoda din **clasa de bază** sau excepții **derivate** din acestea.

Java I/O, NIO, NIO2

- *Pachetul java.io*
 - clase pentru lucrul cu octeti (*InputStream*, *OutputStream*)
 - clase pentru lucrul cu caractere (*Reader*, *Writer*)
 - conversie octeti-caractere (*InputStreamReader*, *OutputStreamWriter*)
 - acces aleator (*RandomAccessFile*)
 - serializarea obiectelor (*ObjectInputStream*, *ObjectOutputStream*)
- *Pachetul Java.util - Utilitare: Scanner*
- *Pachetul java.nio* (Non-blocking I/O)
- *Pachetul java.nio2*

Conceptul de stream (flux de date)

- *Stream (flux de date)* = orice *sursă* sau *consumator* de date care este capabil să producă sau să primească *unități* de date.
- 3 contexte în care apare conceptul de stream: Java IO, Java NIO, Java 8 – Stream (functional programming)

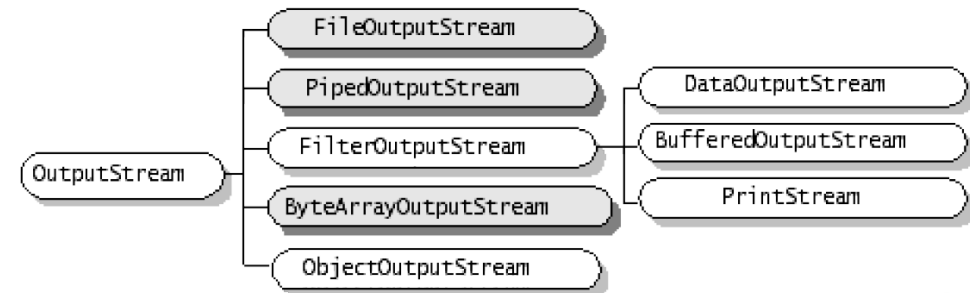
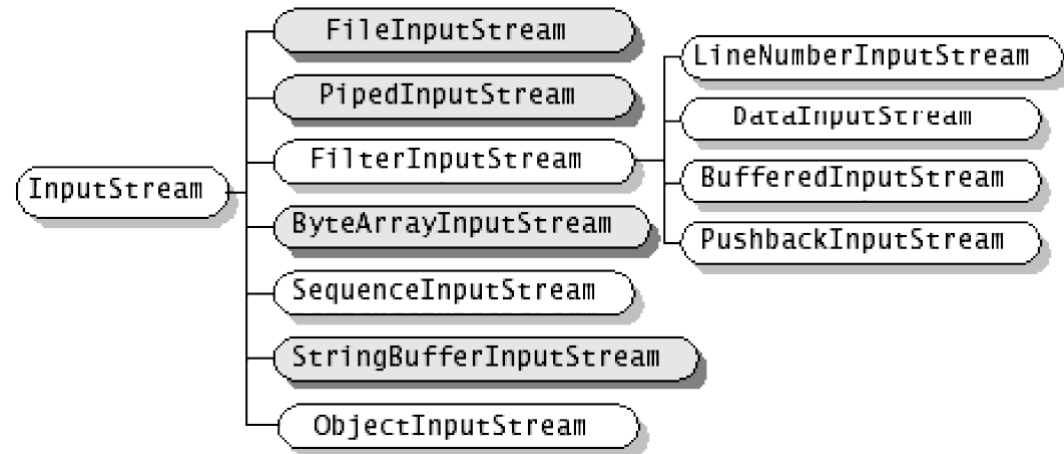


Tipuri de fluxuri de date (data streams)

- Pe octeti (byte) - unitatea de informatie: OCTETUL
- Pe caractere - unitatea de informatie: CHARACTERUL

Fluxuri pe octeti

InputStream, **OutputStream** - Citire/scriere octet cu octet



[Source: java.sun.com]

Input Stream

- Clasa abstracta **InputStream** se situeaza in varful ierarhiei de clase care descriu fluxuri octet **sursa**
- Toate clasele derivate (care nu sunt abstracte) implementeaza metoda **read()**

abstract int

read()

Reads the next byte of data from the input stream.

int

read(byte[] b)

Reads some number of bytes from the input stream and stores them into the buffer array b.

int

read(byte[] b, int off, int len)

Reads up to len bytes of data from the input stream into an array of bytes.

void

reset()

Repositions this stream to the position at the time the mark method was last called on this input stream.

long

skip(long n)

Skips over and discards n bytes of data from this input stream.

Output Stream

- Clasa **OutputStream** se situeaza in varful ierarhiei de clase care descriu fluxuri octet destinatie.
- Pentru fiecare clasa InputStream exista o clasa omolog OutputStream.
- Operatiile din clasa OutputStream sunt operatiile in oglinda ale celor din clasa InputStream: **write(int b)** etc.

void

close()

Closes this output stream and releases any system resources associated with this stream.

void

flush()

Flushes this output stream and forces any buffered output bytes to be written out.

void

write(byte[] b)

Writes b.length bytes from the specified byte array to this output stream.

void

write(byte[] b, int off, int len)

Writes len bytes from the specified byte array starting at offset off to this output stream.

abstract void

write(int b)

Writes the specified byte to this output stream.

Exemplu – FileInputStream/FileOutputStream

```
public static void readWithFileInputStream() {
    FileInputStream in = null;
    FileOutputStream out = null;
    try {
        in = new FileInputStream("Fis1.txt");
        out = new FileOutputStream("Fis2.txt");
        int c;
        while ((c = in.read()) != -1) {
            out.write(c);
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (in != null)
            try {
                in.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        if (out != null)
            try {
                out.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
    }
}
```

```
public static void readWithFileInputStream(){

    try(FileInputStream is=new
FileInputStream("./data/Messages");
        FileOutputStream os =new
FileOutputStream("./data/CopyMessages")){
        int i;
        while((i=is.read())!=-1){
            os.write(i);
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```


Try-With-Resources

```
public static void readBytes_usingTryWithResources() {
    try (FileInputStream in = new FileInputStream("Fis1.txt");
        FileOutputStream out = new FileOutputStream("Fis2.txt"))
    {
        int c;
        while ((c = in.read()) != -1) {
            out.write(c);
        }
    } catch (IOException e) {
        // aici nu avem acces la var in sau out
    }
}
```

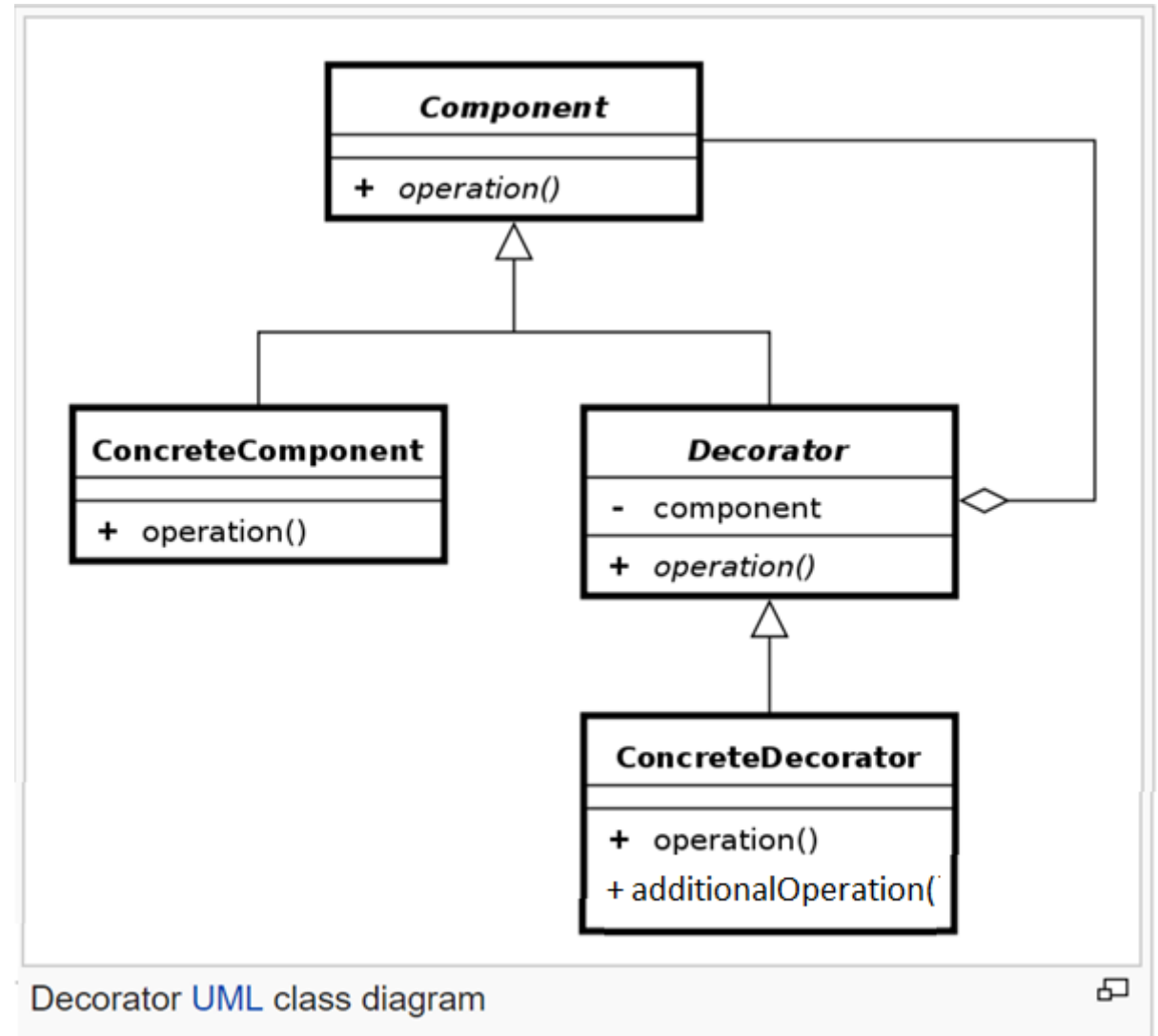
Pentru a folosi *Try-With-Resources* Resursa trebuie sa implementeze interfața **java.lang.AutoCloseable**

Old stye

```
public static void readBytes() {
    FileInputStream in = null; FileOutputStream out = null;
    try {
        in = new FileInputStream("Fis1.txt");
        out = new FileOutputStream("Fis2.txt");
        int c;
        while ((c = in.read()) != -1) {
            out.write(c);
        }
    } catch (IOException e) { e.printStackTrace(); }
    finally {
        if (in != null)
            try {
                in.close();
            } catch (IOException e) { e.printStackTrace(); }
        if (out != null)
            try {
                out.close();
            } catch (IOException e) { e.printStackTrace(); }
    }
}
```

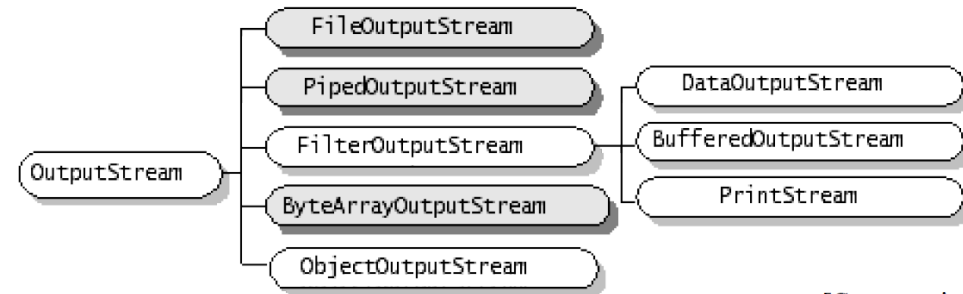
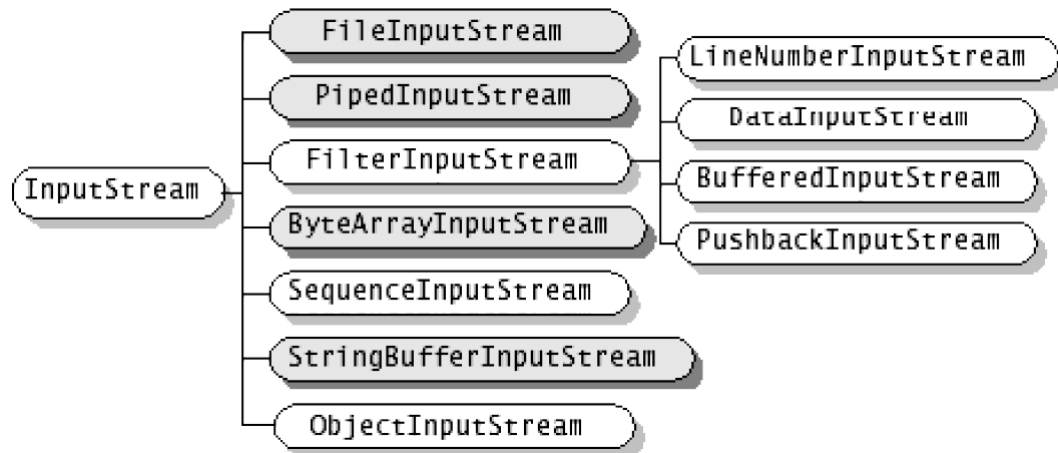
Clase decorator pentru fluxuri

- Problema:
 - Ne dorim sa citim int, String, double.....
 - De exemplu: Avem nevoie de un InputStream care citeste 4 octeti deodata....
- Solutia: **Decorator design pattern**
 - Acest pattern impune ca obiectele care adauga functionalitate (**wrappers**) unui obiect anume sa aibe aceeasi **interfata**.



Clase decorator pentru fluxuri

- Fluxurile *primitive* “știu” să facă efectiv operații de citire/scriere de la/către un “partener” extern (fișier, memorie, fir de execuție, etc.)
- Fluxurile de *filtrare* (*decorators*) “știu” să comunice cu un flux primitiv (sau alt flux de filtrare) **pentru a procesa și oferi datele într-un mod mai complex.**



[Source: java.sun.com]

Clase decorator pentru fluxuri - exemplu

- **Decoratori:**

- Clasele **DataInputStream** și **DataOutputStream** - Citirea/Scrierea tipurilor primitive
- Adauga metodele:
 - **readByte**, **readInt**, **readFloat**, **readBoolean**, si omoloagele lor, **writeByte**, **writeInt** etc.

- Este de notat folosirea concomitentă a **FileOutputStream** și **FileInputStream**.

- Pentru a putea citi datele de tip primitiv folosind metodele **readYyy**, ele trebuie intai salvate folosind metodele **writeYyy**.

Decoratori – Data Output/Input Stream

```
public static void writeFileDataOutputStream()
{
    try (DataOutputStream out = new DataOutputStream(new FileOutputStream("fis.meu"))){
        out.writeFloat(10.0f);
        out.writeChar('\n');
        out.writeInt(5);
        out.writeChar('\n');
        out.writeUTF("Hello");

    } catch(IOException e) {e.printStackTrace(); }
}
```

```
public static void readFileDataInputtStream(){
    try (DataInputStream in = new DataInputStream(new FileInputStream("fis.meu"))){
        System.out.println("Citesc un float: " + in.readFloat());
        System.out.println("Citesc enter: " + in.readChar());
        System.out.println("Citesc un int: " + in.readInt());
        System.out.println("Citesc enter: " + in.readChar());
        System.out.println("Citesc un string: " + in.readUTF());
    } catch(IOException e) { e.printStackTrace();}
}
```

Decoratori — BufferedOutputStream/ BufferedInputStream

```
public static void writeBufferedFileDataInputtStream()  
{  
    try (BufferedOutputStream bout=new BufferedOutputStream(new FileOutputStream("fis.meu"));  
        DataOutputStream out = new DataOutputStream(bout)) {  
        out.writeFloat(10.0f);  
        out.writeChar('\n');  
        out.writeInt(5);  
        out.writeChar('\n');  
        out.writeUTF("Hello");  
    } catch(IOException e) {e.printStackTrace(); }  
}  
public static void readBufferedDataOutputStream(){  
    try (BufferedInputStream bin=new BufferedInputStream(new FileInputStream("fis.meu"));  
        DataInputStream in = new DataInputStream(bin))  
    {  
        System.out.println("Citesc un float: " + in.readFloat());  
        System.out.println("Citesc enter: " + in.readChar());  
        System.out.println("Citesc un int: " + in.readInt());  
        System.out.println("Citesc enter: " + in.readChar());  
        System.out.println("Citesc un string: " + in.readUTF());  
    } catch(IOException e) { e.printStackTrace();}  
}
```

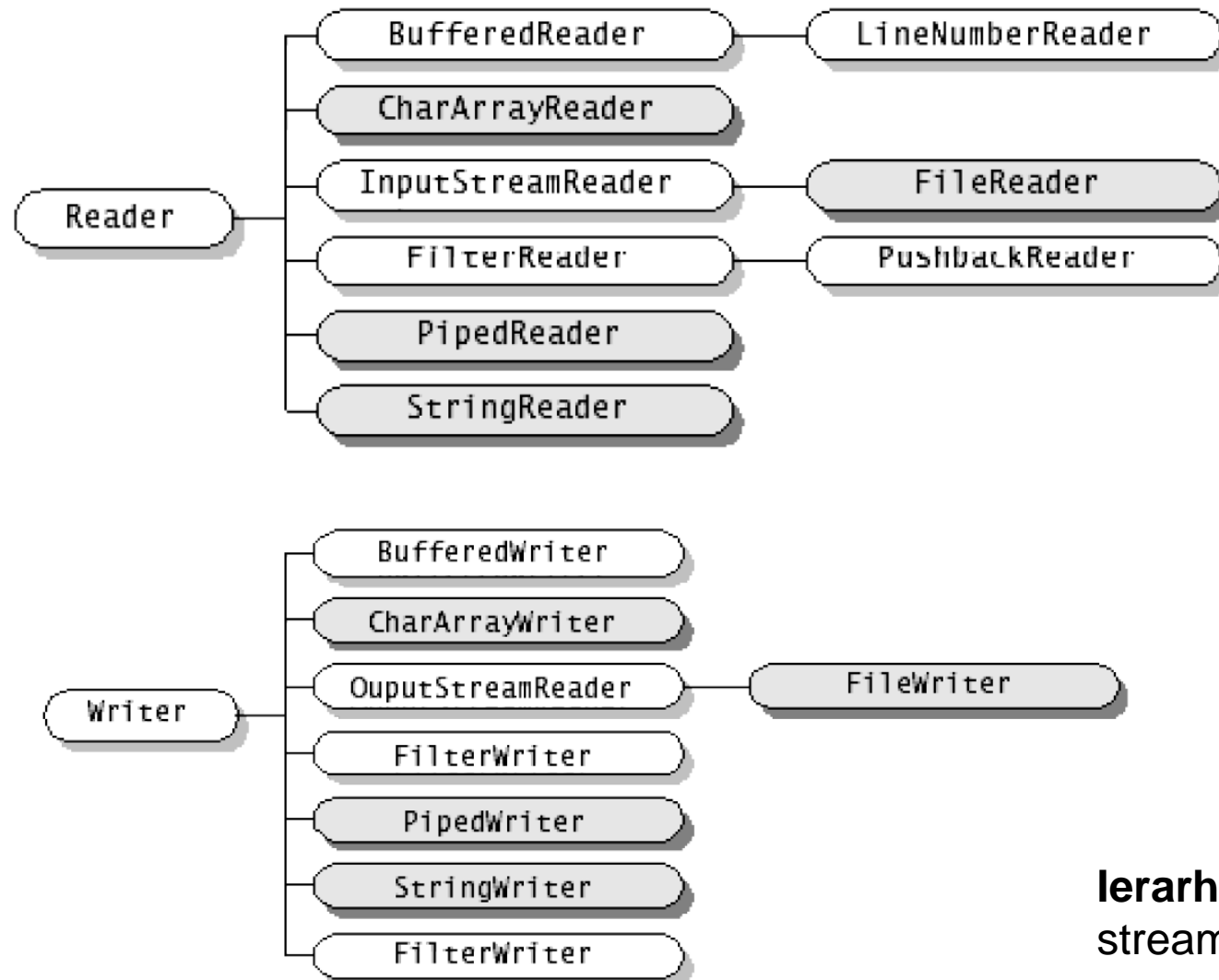
Streamuri standard

- `System.in` de tipul `InputStream`
- `System.out` de tipul `PrintStream`
- `System.err` de tipul `PrintStream`
- Se pot modifica stream-urile asociate folosind metodele: `System.setIn()`, `System.setOut()`, `System.setErr()`,

Fluxuri pe caractere

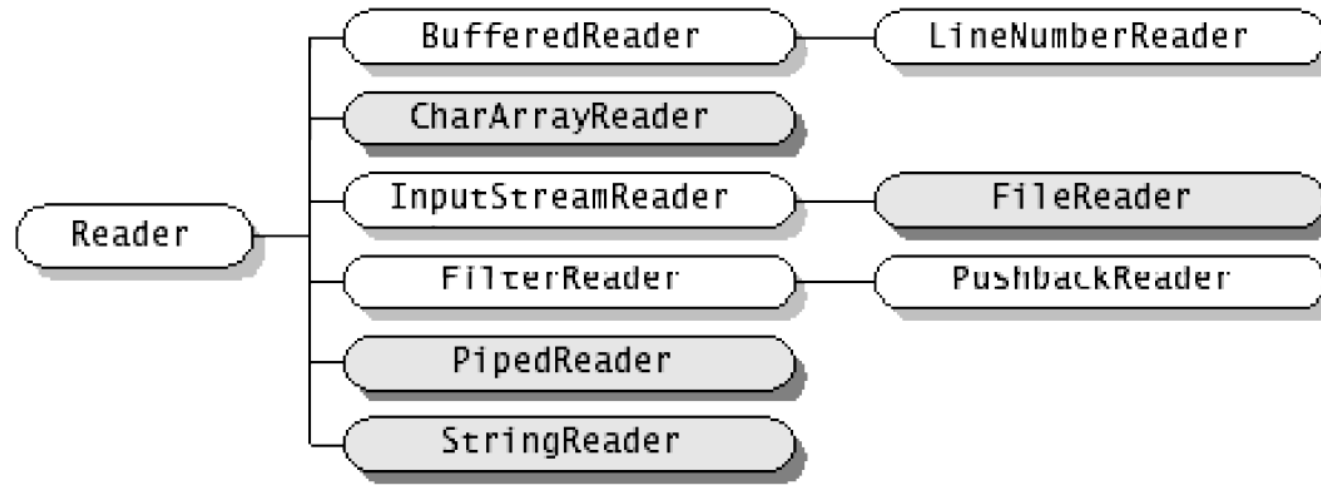
- Odata cu introducerea caracterelor *Unicode* a crescut numarul de octeti necesari pentru a reprezenta un character, de la 1 la 2.
- De asemenea, a aparut notiunea de *codificare* a sirurilor (*encoding*).
- Necesitatea introducerii unei noi perspective asupra stream-urilor, la nivel de **character**.
- In varful ierarhiilor claselor care lucreaza cu caractere se afla Reader si Writer.
- *Reader si Writer ofera primitive asemanatoare cu cele din InputStream/OutputStream, cu diferenta ca este folosit caracterul si nu octetul ca unitate de informatie.*
- Flosesc *decorators*
- Daca dorim sa citim/scriem siruri de caractere trebuie sa folosim Reader si Writer

Fluxuri pe caractere



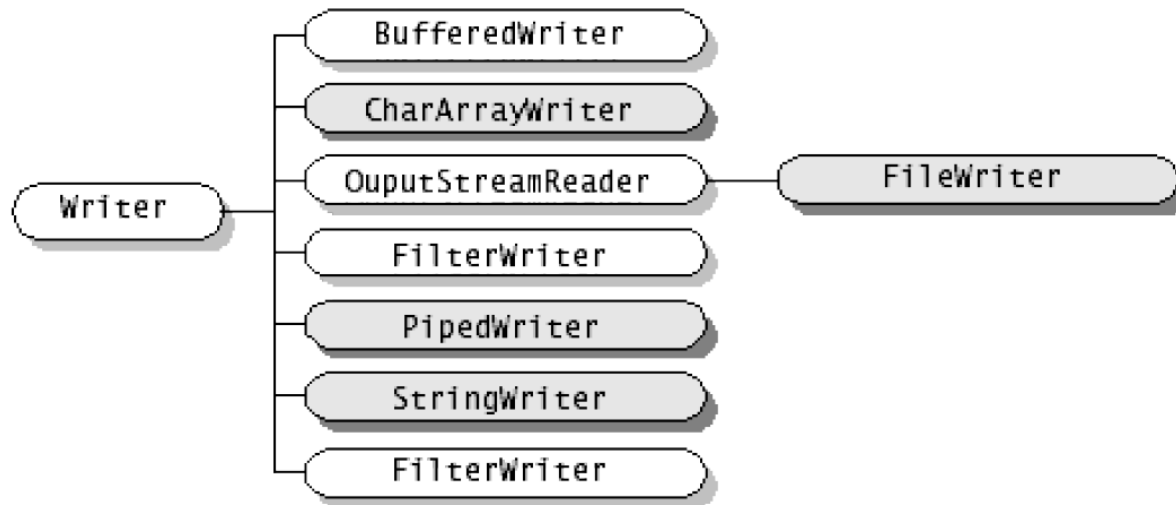
Ierarhia de clase este aproape identica cu cea de la stream-uri octet.

Ierarhia de clase Reader



- **Reader**, the abstract component root in decorator pattern
- **BufferedReader**, etc. the concrete components
- **FilterReader**, the abstract decorator
- **PushbackReader**, concrete decorators

Ierarhia de clase Writer



- **Writer**, the abstract component root in decorator pattern
- **BufferedWriter**, etc. the concrete components
- **FilterWriter**, the abstract decorator
- • No concrete decorators

Octeti - caractere

Operatii	Octeti	Caractere
Lucrul cu fisiere	FileInputStream, FileOutputStream	FileReader, FileWriter
Salvarea in memorie	ByteArrayInputStream, ByteArrayOutputStream	CharArrayReader CharArrayWriter
Folosirea unei zone tampon	BufferedInputStream BufferedOutputStream	BufferedReader BufferedWriter
Tiparire (Formatare)	PrintStream	PrintWriter
Conversie octeti - caractere	InputStreamReader (octeti -> caractere) OutputStreamWriter (caractere -> octeti)	

Fluxuri de caractere. Citirea cu buffer

- Primitiv: `FileReader`;
- Decorator: `BufferedReader`.

```
public static List<Student> readStudentsFromFile()
{
    ArrayList<Student> students = new ArrayList<>();
    try (BufferedReader br = new BufferedReader(new FileReader("studenti.txt"))) {
        String line;
        while((line = br.readLine()) != null)
        {
            String[] fields = line.split("\\|");
            if(fields.length != 3){
                throw new Exception("Fisier corrupt!");
            }
            Student s = new Student(Integer.parseInt(fields[0]), fields[1], Float.parseFloat(fields[2]));
            students.add(s);
        }

    } catch (FileNotFoundException e) { System.out.println("Fisierul nu a fost gasit!"); }
    catch (IOException e) { System.out.println("Eroare la citire"); }
    catch (Exception e) { e.printStackTrace(); }
    return students;
}
```

Citirea de la tastatura - cu buffer

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

Fluxuri de caractere. Scrierea cu buffer

- Primitiv: `FileWriter`;
- Decorator: `BufferWriter`.

```
public static void writeStudentsInFile(List<Student> students) {  
    try (BufferedWriter bf = new BufferedWriter(new FileWriter("studenti.txt"))) {  
        for(Student student:students){  
            bf.write(student.toString());  
            bf.write('\n');  
        }  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

Clasa Scanner

```
import java.util.Scanner;
```

Clasa **Scanner** contine metode ce permit citirea tipurilor primitive de la tastatura (sau alt flux de date):

- `nextInt():int`
- `nextDouble():double`
- `nextFloat():Float`
- `nextLine():String`
- ...
- `hasNextInt():boolean`
- `hasNextDouble():boolean`
- `hasNextFloat():boolean`
- ...

Citirea din fisier cu Scanner

```
import java.util.Scanner;

try (Scanner scanner = new Scanner(new FileInputStream("studenti.txt"))) {
    String line;
    while(scanner.hasNextLine())
    {
        line=scanner.nextLine();
        String[] fields = line.split("\\|");
        if(fields.length != 3){
            throw new Exception("Fisier corrupt!");
        }
        Student s = new Student(Integer.parseInt(fields[0]), fields[1], Float.parseFloat(fields[2]));
        students.add(s);
    }
}
```

Citirea de la tastatura cu Scanner

```
Scanner scanner = new Scanner(System.in);  
String nume;  
int id;  
float media;  
id=scanner.nextInt();  
scanner.nextLine();  
nume=scanner.nextLine();  
media=scanner.nextFloat();  
Student s = new Student(id,nume,media);
```

Clasa File

- Reprezinta numele unui fisier (nu continutul acestuia).
- Permite scrierea de operatii cu fisiere (creare, stergere, redenumire, etc.) intr-un mod independent de platforma (sistemul de operare).
 - `File(nume:String)` //nume reprezinta calea catre un fisier sau director
 - `getName():String`
 - `getAbsolutePath():String`
 - `isFile():boolean`
 - `isDirectory():boolean`
 - `exists():boolean`
 - `delete():boolean`
 - ...

Serializarea obiectelor

ObjectInputStream , ObjectOutputStream

- Procesul de scriere/citire a obiectelor din/in fisier/suport extern.
- Un obiect **persistent** (**serializabil**) este un obiect ce poate fi scris in fisier/suport extern, respectiv citit din fisier sau suport extern
- Declararea claselor a caror obiecte sunt serializabile se face cu ajutorul interfetei **Serializable** (pachetul **java.io**)

```
public class Student implements Serializable{ ... }
```

- Interfata Serializable nu contine nici o metoda.

```
try (ObjectOutputStream out= new ObjectOutputStream(new  
FileOutputStream("studentSer.txt"))){  
    Student stud=new Student(1,"Popescu Ioan", 7.9f);  
    out.writeObject(stud);  
}  
catch (IOException e) {  
    e.printStackTrace();  
}
```

- Se salveaza pe disc starea obiectului **stud** (valorile variabilelor membru).

Serializarea obiectelor

- Obiectele referite de un obiect serializabil, trebuie sa fie la randul lor serializabile
- Atributele **static** ale unei clase serializabile nu sunt salvate in fisier/suport extern.
- Un obiect de la o anumita referinta este salvat o singura data pe acelasi stream
- Metoda `in.readObject():Object`
 1. Se citeste obiectul de pe stream
 2. Se identifica tipul obiectului
 3. Se initializeaza datele membre nestatice octet cu octet(fara a apela un constructor) si se returneaza obiectul creat.
- Metoda `out.writeObject(Object)`

Se salveaza valorile atributelor nestatice si informatii care ajuta masina virtuala Java sa reconstruiasca obiectul.

Un obiect de la o anumita referinta este salvat o singura data pe acelasi stream.

Serializarea obiectelor - serialVersionUID

- ```
public class Student implements Serializable{
 private String nume;
 private double media;

 //...
}
```

- Scenariu:

1. Se serializeaza obiecte de tip Student.
2. Se modifica clasa Student (se adauga/sterge attribute/metode).
3. Se doreste deserializarea obiectelor salvate.

- ```
public class Student implements Serializable{  
[orice modif acces] static final long serialVersionUID = 1L;  
    private String nume;  
    private double media;  
    private int grupa;  
  
    //...  
}
```

- Atributele noi adaugate vor fi initializate cu valorile implicite corespunzatoare tipului lor.

Serializarea obiectelor -transient

- Exista cazuri cand nu se doreste salvarea valorilor unor attribute (*ex. parole, descriptori de fisiere, etc.*)
- Aceste attribute se declara folosind cuvantul **transient**:
- ```
public class Student implements Serializable{
 private String nume;
 private double media;
 private transient String parola;
 //...
}
```
- La citire, attributele declarate cu transient vor fi initializate cu valoarea implicita corespunzatoare tipului lor.

# Java NIO - Non-blocking IO sau New IO

- Incepand cu versiunea 1.4 – Java NIO sunt o alternativă la Java IO API si la Java Networking API
- Operatiile de citire/scriere sunt mai performante deoarece folosesc structuri asemanatoare cu cele folosite de sistemul de operare: **canal de comunicare (eng. *channel*)**, **zona tampon (eng. *buffer*)**
- **Au la bază urmatoarele 3 concepte:**
  - Channels
  - Buffers
  - Selectors



# Java NIO – Channels and Buffers

- Datele sunt întotdeauna citite dintr-un channel într-un buffer sau scrise de la un buffer într-un channel.
- Față de stream-ri, channel-rile sunt *bidirectionale*. Putem scrie și citi ...
- Permit execuția **asincronă** a operațiilor de IO
- Clase:
  - **FileChannel** - canal de comunicare pentru fișiere.
  - **MappedByteBuffer** - manipularea fișierelor foarte mari
  - **FileLock** - permite sincronizarea accesului la un fișier
  - Clasele **FileInputStream**, **FileOutputStream** și **RandomAccessFile** au metode care returnează un obiect de tip **FileChannel**.
- *Clasele din pachetul java.io au fost rescrise folosind clasele din pachetul java.nio pentru a îmbunătăți performanța lor!!!!*

# Java NIO.2

## Path

- Interfața `java.nio.file.Path` sau `Path` pentru scurt este punctul de pornire pentru lucrul cu API-ul NIO.2.
- `Path` este un înlocuitor direct pentru `java.io.File`:
  - obiectele `File` și `Path` se pot referi la un fișier sau la un director.
  - de asemenea, se pot referi la o cale absolută sau o cale relativă în cadrul sistemului de fișiere.
- Spre deosebire de clasa `File`, interfața `Path` conține suport pentru “*symbolic links*”. O **legătură simbolică** este un fișier special în cadrul unui sistem de operare care servește ca referință sau pointer la un alt fișier sau director. În general, legăturile simbolice sunt transparente pentru utilizator,
- API-ul NIO.2 include suport complet pentru crearea, detectarea și navigarea legăturilor simbolice în interiorul sistemului de fișiere.

# Crearea instanțelor folosind Factory

## De ce este Path o interfață?

- Când obțineți un obiect Path, JVM oferă un obiect care, spre deosebire de `java.io.File`, gestionează în mod transparent detaliile specifice sistemului de fișiere pentru platforma curentă.
- Java NIO.2 folosește șablonul Factory pentru crearea obiectelor de tipul Path, prin intermediul clasei `Paths` ce conține metode statice.
- Dacă nu folosim șablonul Factory, pentru a crea o instanță, are trebui să stim care a fost sistemul de fișiere care a stat la baza creării instanței.
- Avantajul utilizării șablonului Factory este că puteți scrie același cod care rulează apoi pe platforme diferite.

# Clase Helper pentru obiecte Path

- **NIO.2** include, de asemenea, clase helper cum ar fi **java.nio.file.Files**, al căror scop principal este de a opera pe instanțe ale obiectelor Path.
- Clasele helper sunt asemănătoare cu clasele factory în sensul că ele sunt adesea compuse în primul rând din metode statice care operează pe o anumită clasă.
- Ele diferă de clasele helper prin faptul că se concentrează asupra *manipulării obiectelor claselor existente*, în timp ce clasele factory se concentrează în primul rând pe crearea de obiecte.

# Java NIO2

```
public static void readWriteStuds()
{
 Path path = Paths.get("./src/data/Studs.txt");
 Stream<String> lines; //Stream - A sequence of elements supporting sequential and parallel
 // aggregate operations.

 try {
 lines = Files.lines(path);
 lines.forEach(s -> System.out.println(s));
 } catch (IOException e) {
 System.out.println(e.getMessage());
 }

 try (BufferedWriter bufferedWriter = Files.newBufferedWriter(path, StandardOpenOption.APPEND)) {
 bufferedWriter.write(
 "Linie noua");
 bufferedWriter.newLine();
 } catch (IOException e) { e.printStackTrace() }
}
```

# Cursul următor

- Java 8 features