# Metode Avansate de Programare
# Excepții. I/O în Java. JUnit

Arthur Molnar
*arthur.molnar@ubbcluj.ro*

Universitatea Babeș-Bolyai

2023

# Overview

1. Excepții
2. I/O în Java
   - Serializarea
3. JUnit

# Excepții I

- Excepții - o situație anormală care are loc în cadrul programului în timpul execuției sale.

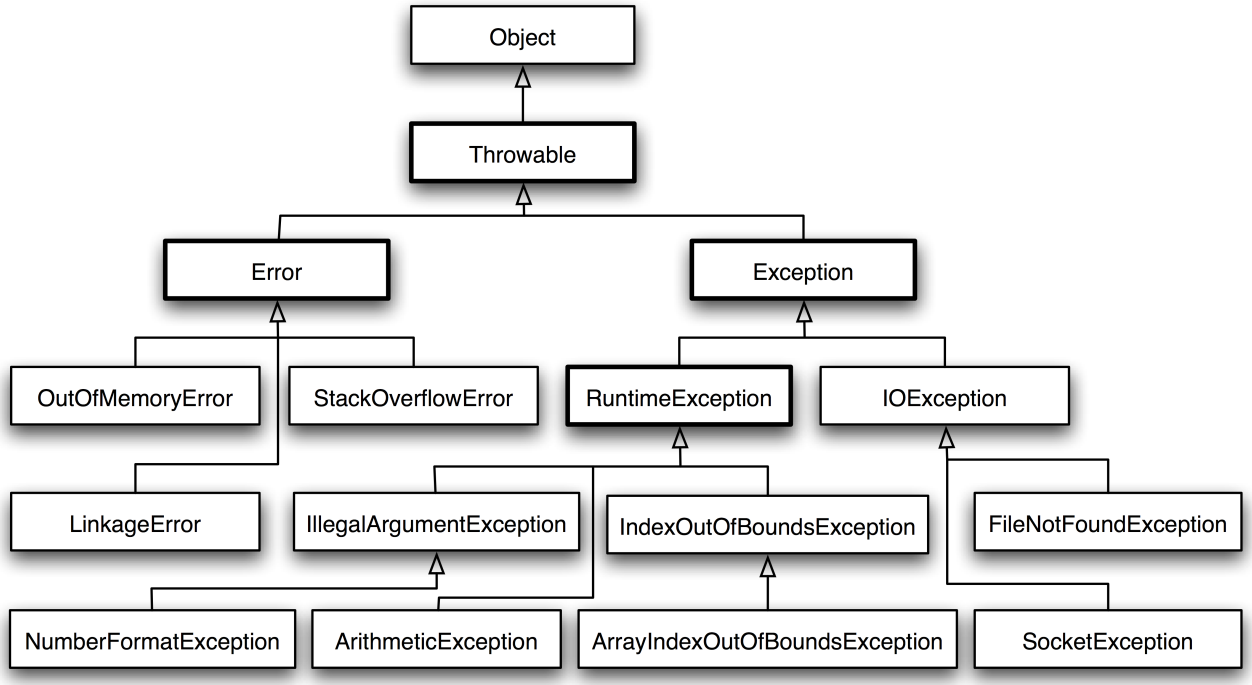


Figure: Sursa figurii: ProTech

# Excepții II

- O *Eroare* (`java.lang.Error`) "indică o problemă gravă pe care o aplicație uzuală nu ar trebui să încerce să o trateze. Cele mai multe astfel de erori reprezintă indică condiții anormale" (API-ul Java).
- O *Excepție* (`java.lang.Exception`) "indică apariția unei condiții pe care o aplicație uzuală ar trebui să o trateze". (API-ul Java).
- `java.lang.Exception` și subclasele care nu sunt derivate din clasa `java.lang.RuntimeException` se numesc *checked exceptions* (eng.).
- La momentul compilării programului se verifică faptul că aceste excepții sunt declarate în clauza `throws` a metodei sau constructorului care le creează și aruncă.

# Excepții III

- `java.lang.Error`, `java.lang.RuntimeException` și subclasele acestora se numesc *unchecked exceptions* (eng).
- Mașina vrituală Java (JVM) aruncă `java.lang.Error` sau una din subclasele acesteia în momentul în care execuția aplicației nu mai poate fi continuată.
- Excepțiile de tipul *unchecked exceptions* pot fi create și aruncate fără a fi declarate în clauzele *throws* corespunzătoare.
- În general excepțiile de tipul *checked exception* sunt utilizate pentru a semnala apariția unei situații excepționale în cadrul programului (ex. `java.io.FileNotFoundException`) în timp ce excepțiile de tipul *unchecked exception* semnalează apariția unui defect (ex: `java.lang.NullPointerException`), apariția acestora din urmă trebuind eliminate în procesul de dezvoltare și testare al aplicației

# Throw și throws

- `throw` eset utilizat pentru a arunca o excepție în cadrul unei metode.
- Excepția trebuie să fie o instanță a clasei `java.lang.Throwable`
- `throws` este utilizat în semnătura unei metode pentru a indica tipurile de excepții (*checked exceptions*) pe care metoda le poate arunca.
- Apelantul metodei va trebui să trateze excepția folosind un bloc `try-catc` sau să o declare în cadrul semnăturii proprii prin utilizarea mecanismului `throws`.

# try-catch-finally

```
try
{
    method1();
    method2(); // arunca exceptie
    method3(); // nu se mai executa in cazul
               // aruncarii unei exceptii
}
catch (Type_Exception1 ex1){ // trateaza ex1 }
catch (Type_Exception1 ex2){ // trateaza ex2 }
catch (Type_Exception3 | Type_Exception4 ex3){
    // trateaza ex3
}
finally{
    // codul din blocul finally este tot timpul executat
}
```

# Definirea propriilor tipuri de excepții

- Excepțiile deja definite în limbajul Java acoperă aproape toate tipurile de situații care pot apărea într-un program
- În anumite cazuri însă, avem nevoie să definim propriile tipuri de excepție (motivele țin des și de arhitectura sistemului și lizibilitatea codului)
- De regulă, excepțiile definite de noi vor fi subclase ale `java.lang.Exception`

# java.lang.Exception

- Constructori:
  - `Exception()`
  - `Exception(String message)`
  - `Exception(String message, Throwable cause)`
  - `Exception(Throwable cause)`
- Metode (selecție):
  - `getCause(): Throwable`
  - `getMessage(): String`
  - `printStackTrace()`
  - `printStackTrace(PrintStream s)`

# Excepțiile și mecanismul de moștenire

- Constructorul clasei derivate trebuie să specifice toate excepțiile specificate în constructorul clasei de bază, putând adăuga propriile tipuri de excepție.
- Metodele suprascrise (eng. *overriden*) pot arunca doar acele tipuri de excepție care au fost specificate în clasa de bază sau subclase ale acestora.
- Regula de mai sus se aplică și în cazul metodelor declarate în cadrul interfețelor.

# Exemple

## Exemplu

`lecture.examples.lecture4.exceptions.Examples.java`[a]

[a] acesta este un link, toate exemplele sunt la adresa https://github.com/cs-ubbcluj-ro/MAP

# I/O în Java

- Pachetul `java.io` oferă:
  - Clase care funcționează cu date binare: `java.io.InputStream`, `java.io.OutputStream`.
  - Clase care funcționează cu date de tip caracter: `java.io.Reader`, `java.io.Writer`.
  - Clase pentru conversia între date binare și date de tip caracter: `java.io.InputStreamReader`, `java.io.OutputStreamWriter`.
  - Clase utilizate pentru serializarea obiectelor: `java.io.ObjectInputStream`, `java.io.ObjectOutputStream`.

# Fluxuri - recapitulare (eng. *Streams*)

- Un flux reprezintă o abstractizare pentru primirea/trimiterea de date într-o situație de intrare/ieșire.
- Fluxul este utilizat doar pentru transmiterea datelor.
- Implementările concrete primesc date de la o sursă bine definită (ex. tastatură, memorie, fișier, zonă de memorie partajată) și o trimit spre o destinație bine definită (monitor, memorie, fișier, etc.)
- Fluxurile sunt în general asociate unei surse sau destinații fizice precum un fișier de pe disc, tastatura, sau consola.
- Mecanismul de fluxuri oferă sprijin pentru lucrul cu octeți, caractere, tipuri primitive de date și obiecte.

# java.io.InputStream I

- Clasa abstractă `java.io.InputStream` este clasa de bază a tuturor claselor ce reprezintă un flux de intrare care lucrează cu octeți.
- Toate clasele non-abstracte derivate implementează metoda `read()`.

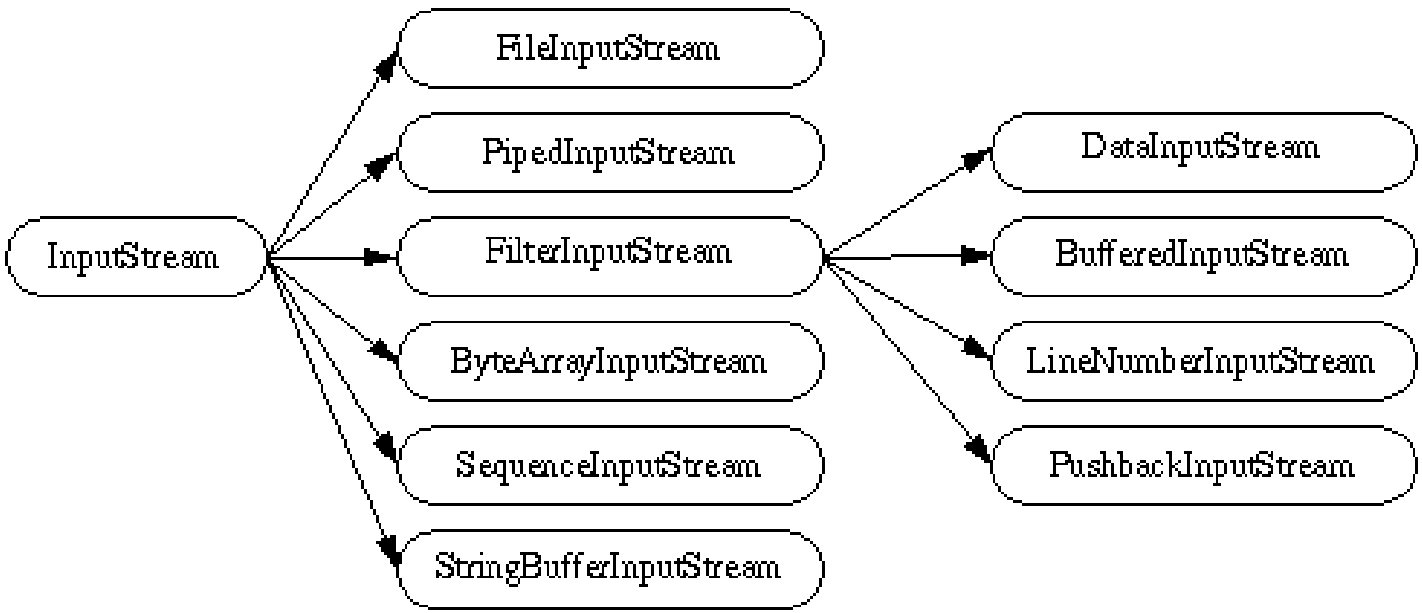


Figure: Sursa figurii: Privire de ansamblu a fluxurilor de intrare și ieșire

# `java.io.InputStream` II

- `InputStream` definește metode pentru:
  - Citirea octeților sau a tablourilor de octeți
  - Marcarea locațiilor în cadrul unui flux
  - Sărirea pentru unii octeți din fluxul de intrare
  - Resetarea poziției curente
- Un flux de intrare este deschis în mod automat la creare
- Fluxurile pot fi închise explicit cu metoda `close()`, sau pot fi închise în mod automat de componenta *garbage collector*. Este recomandat să închidem fluxurile odată ce nu le mai utilizăm.

# java.io.ObjectOutputStream

- Clasa abstractă `java.io.ObjectOutputStream` este clasa de bază a tuturor claselor ce reprezintă un flux de ieșire care lucrează cu octeți.
- Toate clasele non-abstracte derivate implementează metoda `write()`.

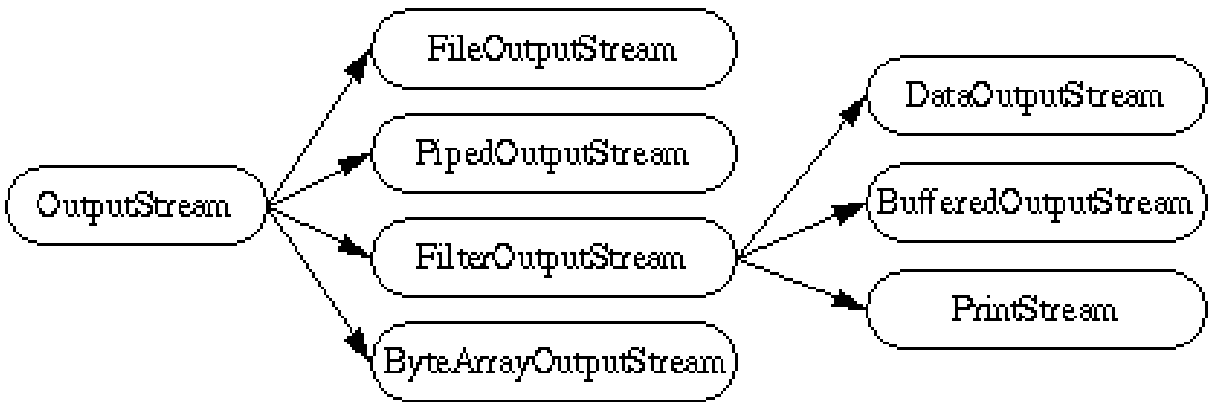


Figure: Sursa figurii: Privire de ansamblu a fluxurilor de intrare și ieșire

# java.io.ObjectOutputStream II

- `OutputStream` definește metode pentru scrierea octaților sau a tablourilor de octeți.
- Un flux de ieșire este deschis în mod automat la creare
- Fluxurile pot fi închise explicit cu metoda `close()`, sau pot fi închise în mod automat de componenta *garbage collector*. Este recomandat să închidem fluxurile odată ce nu le mai utilizăm.

# java.io.Reader și java.io.Writer

- `Reader` și `Writer` sunt clase abstracte pentru citirea/scrierea fluxurilor bazate pe caractere.
- Metodele importante care trebuie implementate în cadrul claselor derivate:
  - `read()`
  - `write()`

## Exemplu

`lecture.examples.lecture4.streams.IOReaderWriter.java`[a]

[a] acesta este un link, toate exemplele sunt la adresa https://github.com/cs-ubbcluj-ro/MAP

# Clase disponibile

| **Operație** | **Octeți** | **Caractere** |
|---|---|---|
| Lucrează cu fișiere | `FileInputStream`, `FilePutputStream` | `FileReader`, `FileWriter` |
| Lucrează cu memorie | `ByteArrayInputStream`, `ByteArrayOutputStream` | `CharArrayReader`, `CharArrayWriter` |
| Operații cu buffer | `BufferedInputStream`, `BufferedOutputStream` | `BufferedReader`, `BufferedWriter` |
| Formatare | `PrintStream` | `PrintWriter` |
| Conversie între octeți și caractere | `InputStreamReader` (byte ->char) `OutputStreamWriter` (char ->byte) | |

Table: Detalii implementări colecții

# Fluxuri standard

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

# Fluxuri cu memorie tampon (eng. *buffered streams*) I

- Fluxurile cu memorie tampon citesc datele dintr-o zonă de memorie numită zonă tampon (un tablou de octeți menținut pe plan intern) iar API-ul nativ de intrare este apelat doar în momentul în care această zonă se golește.
- La citire, datele sunt aduse din zona de memorie tampon, și nu din fluxul sursă. Atunci când zona de memorie nu mai are date, aceasta se reumple citind date noi din fluxul de intrare.
- Fluxurile de ieșire care utilizează memorie tampon scriu datele într-o zonă de memorie intermediară, iar API-ul nativ de ieșire se apelează doar când această zonă de memorie se umple.

# Fluxuri cu memorie tampon (eng. *buffered streams*) II

- Pentru fluxurile de ieșire este disponibilă operațiunea `flush()`, care cauzează scrierea datelor prin API-ul de ieșire.
- Operațiile de citire/scriere sunt mai eficient, mai ales în condițiile în care citirea/scrierea unei cantități mai mare de informații (sute, mii de octeți) este la fel de rapidă ca citirea unui singur octet/caracter (ex. hard-discuri, SSD)

## Exemplu

`lecture.examples.lecture4.streams.Buffered.java`[a]

[a] acesta este un link, toate exemplele sunt la adresa https://github.com/cs-ubbcluj-ro/MAP

# Citirea de la intrarea standard

```
BufferedReader br = new BufferedReader(new
                        InputStreamReader(System.in)));
// sau
Scanner input=new Scanner(System.in);
```

# Serializarea I

- Mecanismul de serializare (eng. *serialization*) permite reprezentarea unui obiect sub forma unei secvențe de octeți, prin conversia la un flux de octeți.
- Procesul opus se numește deserializare; în acesta, fluxul de octeți este utilizat pentru a recrea obiectul Java în memoria mașinii virtuale.
- Serializarea este un mecanism de persistare a obiectului.
- Fluxul de octeți creat este independent de platformă. Astfel, obiectele serializate pe o platformă pot fi deserializate pe orice platformă pe care rulează o mașină virtuală Java.
- Un obiect *serializabil* este un obiect care poate fi serializat, deci, există și obiecte care **nu** sunt serializabile.

# Serializarea II

- Clasele care dorim să suporte mecanismul de serializare vor implementa interfața `java.io.Serializable`
- Acestă interfață nu conține nici o metodă (o numim interfață marker)
- Obiectele referite în cadrul unui object serializabil trebuie să fie la rândul lor serializabile
- Câmpurile statice ale unei clase serializabile sunt ignorate în cadrul serializării
- Metode importante **din clasele care implementează interfața de serializare**:
  - `readObject(): Object`
  - `writeObject(Object)`
- Mai multe detalii sunt disponibile aici `https://www.oracle.com/technical-resources/articles/java/serializationapi.html`

# Serializarea III

- Considerați următoarea problemă:
  - Obiectele unei clase sunt serializate.
  - Clasa este apoi modificată (ex. unele câmpuri șterse, altele se adaugă).
  - Vrem să deserializăm obiectele serializate.
- Pentru a preveni problemele cauzate în acest scenariu, fiecare clasă serializabilă va avea asociată o versiune stocată în câmpul `serialVersionUID`.
- Valoarea acestui câmp este verificată în momentul deserializării pentru a verifica faptul că veriunea clasei utilizată în momentul serializării este compatibilă cu versiunea clasei din momentul deserializării.
- O excepție de tipul `java.io.InvalidClassException` este aruncată dacă clasa obiectului serializat are o altă valoare a câmpului `serialVersionUID` față de obiectul deserializat.

# Serializarea IV

- Există cazuri în care nu se dorește serializarea anumitor câmpuri ale clasei (ex. parole, numere/CVV carduri bancare)
- Aceste câmpuri pot fi declarate folosind cuvântul cheie `transient`.
- În momentul deserializării, câmpurile declarate ca find `transient` sunt inițializate cu valorile implicite tipului lor.

## Exemplu

`lecture.examples.lecture4.streams.Serialization.java`[a]

[a] acesta este un link, toate exemplele sunt la adresa https://github.com/cs-ubbcluj-ro/MAP

# JUnit I

- `JUnit` este un cadrul de aplicație (eng. *framework*) gratuit pentru testarea unitară (eng. *unit testing*) automată.
- `JUnit` reprezintă o instanță a arhitecturii `xUnit` pentru cadre de aplicație în domeniul testării automate.
- Un *test unitar* (eng. *unit test*) este scris pentru a verifica un anumit comportament sau stare. Scopul este de a testa o secțiune mică de cod (ex. o metodă sau o clasă).
- Procentajul de cod care este executat în momentul în care se rulează toate testele unitare automate se numește acoperirea codului (eng. *test coverage*).
- De regulă, testele unitare se păstrează într-un pachet separat de pachetele aplicației, deseori numit `test`.

# JUnit II

- Un test `JUnit` este o metodă inclusă într-o clasă utilizată exclusiv în scop de testare. Aceasta este o *clasă de testare*.
- Pentru a stabili că o metodă este o metodă de test, o adnotăm cu `@Test`.
- De regulă, numele claselor de test se termină cu *Test*.
- Mai multe clase de test se pot combina într-o suită de testare (eng. *test suite*).
- Când se rulează o suită de testare, toate clasele de test incluse se rulează în ordinea specificată.

- Configurarea bibliotecilor de testare în IntelliJ