METODE AVANSATE DE PROGRAMARE

Conf.univ.dr. Ana Cristina DĂSCĂLESCU

Temtică curs 8

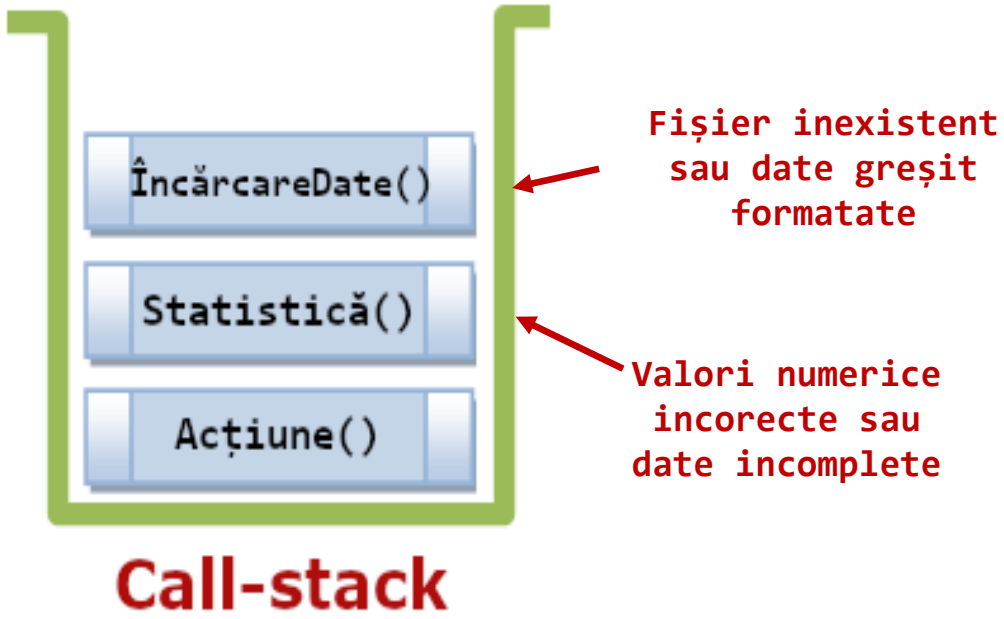
- Mecanismul de tartare al excepțiilor
- Controlul moștenirii
- Fluxuri de intrare/ieșire

Call stack

- **Exemplu:** Aplicație cu o interfață grafică pentru realizarea unei statistici

  - se apelează o metodă "Acțiune",
  - se apelează o metodă "Statistică" dintr-o altă clasă
  - se apelează o metodă "ÎncărcareDate“ pentru a încărca datele dintr-un fișier text.

- **Posibile excepții**
  - calea fișierului cu datele persoanelor este greșită sau fișierul nu există
  - unele persoane au datele eronate în fișier



CALL STACK

- O excepție, trebuie semnalată utilizatorului în interfața grafică, adică trebuie să aibă loc o **propagare a excepției**, fără a bloca funcționalitatea aplicației.

✓În limbajul Java, există un mecanism eficient de tratare a excepțiilor.

✓Practic, o excepție este un obiect care încapsulează detalii despre excepția respectivă, precum metoda în care a apărut, metodele din call-stack afectate, o descriere a sa etc.

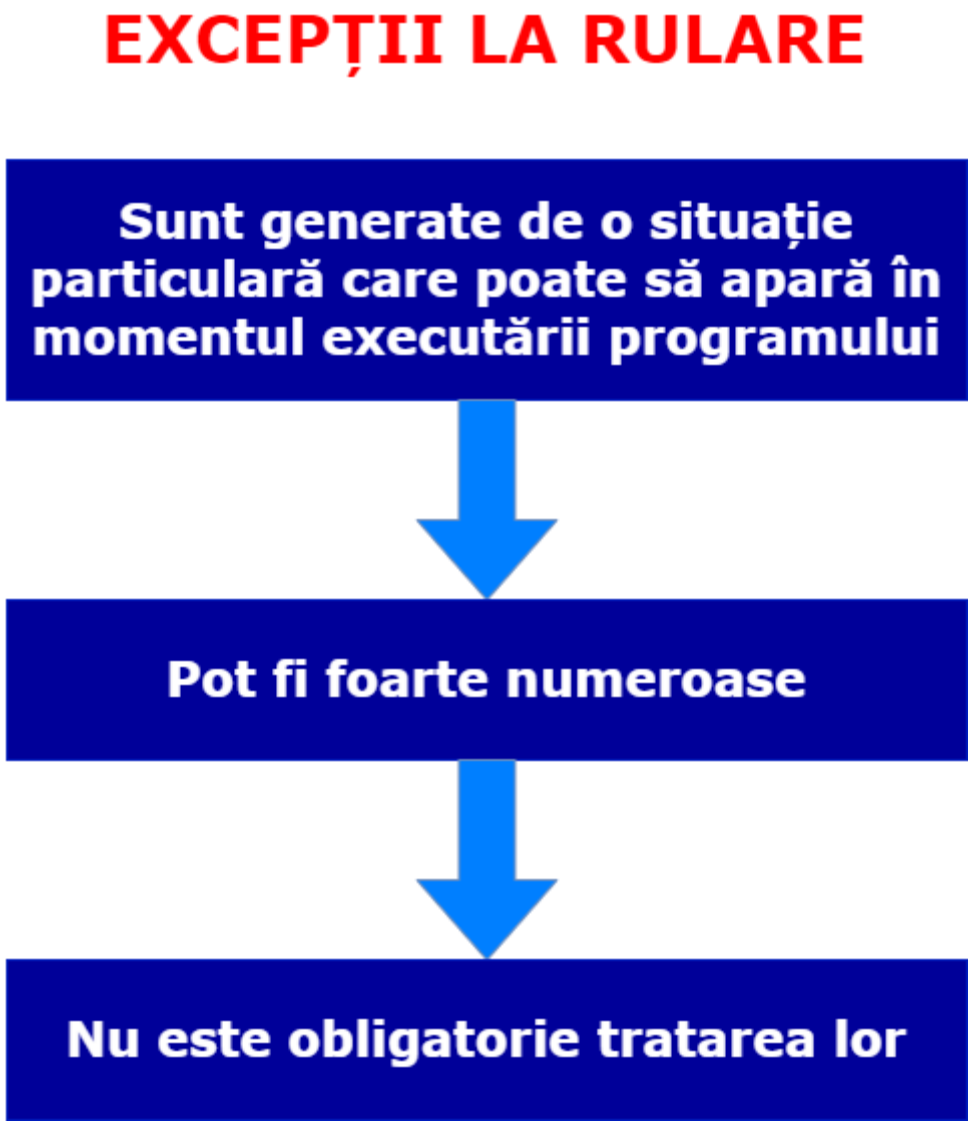
Tipuri de excepții

ERORI

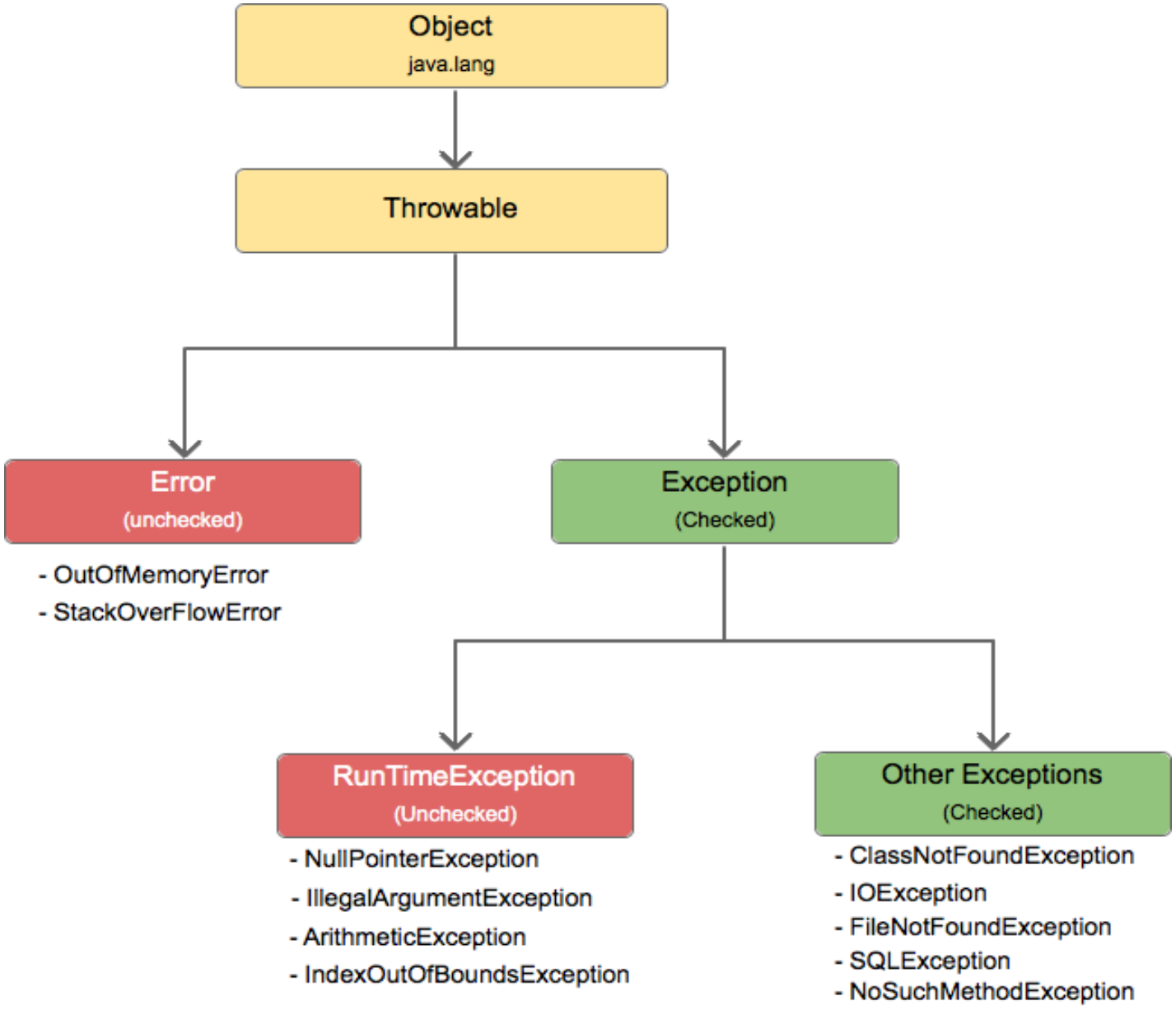
- Sunt generate de hardware sau de Java Virtual Machine
- Nu pot fi anticipate
- Nu este obligatorie tratarea lor

Tipuri de excepții

Tipuri de excepții



Ierarhia de clase pentru tratarea excepțiilor



Exemple de excepții uzuale

- **IOException** – apare în operațiile de intrare/ieșire (de exemplu, citirea datelor dintr-un fișier). O subclasă a clasei **IOException** este **FileNotFoundException,** generată în cazul încercării de deschidere a unui fișier inexistent

```
FileInputStream fin = new FileInputStream("Exemple.in");
```

- **NullPointerException** – folosirea unei referințe cu valoarea **null** pentru accesarea unui membru public sau default dintr-o clasă

```
Persoana ob = null;
ob.getVarsta();
```

- **ArrayIndexOutOfBoundsException** – folosirea unui index incorect, respectiv negativ sau strict mai mare decât dimensiunea fizică a unui tablou - 1;

```
int v[] = {1, 2, 3, 4};
System.out.println(v[4]);
```

Exemple de excepții uzuale

- **`ArithmeticException`** – operații aritmetice nepermise, precum împărțirea unui număr întreg la 0

- **`NumberFormatException`** – conversie a unui String într-un tip de date primitiv din cadrul metodelor **`parseTipPrimitiv`** ale claselor wrapper

```
Float.parseFloat(4,236);
Integer.parseInt("1326589741236");
```

- **`ClassCastException`** – apare la conversia unei referințe către un alt tip de date incompatibil

- **`SQLException`** - excepții care apar la interogarea serverelor de baze de date

Mecanismul Java pentru manipularea excepțiilor

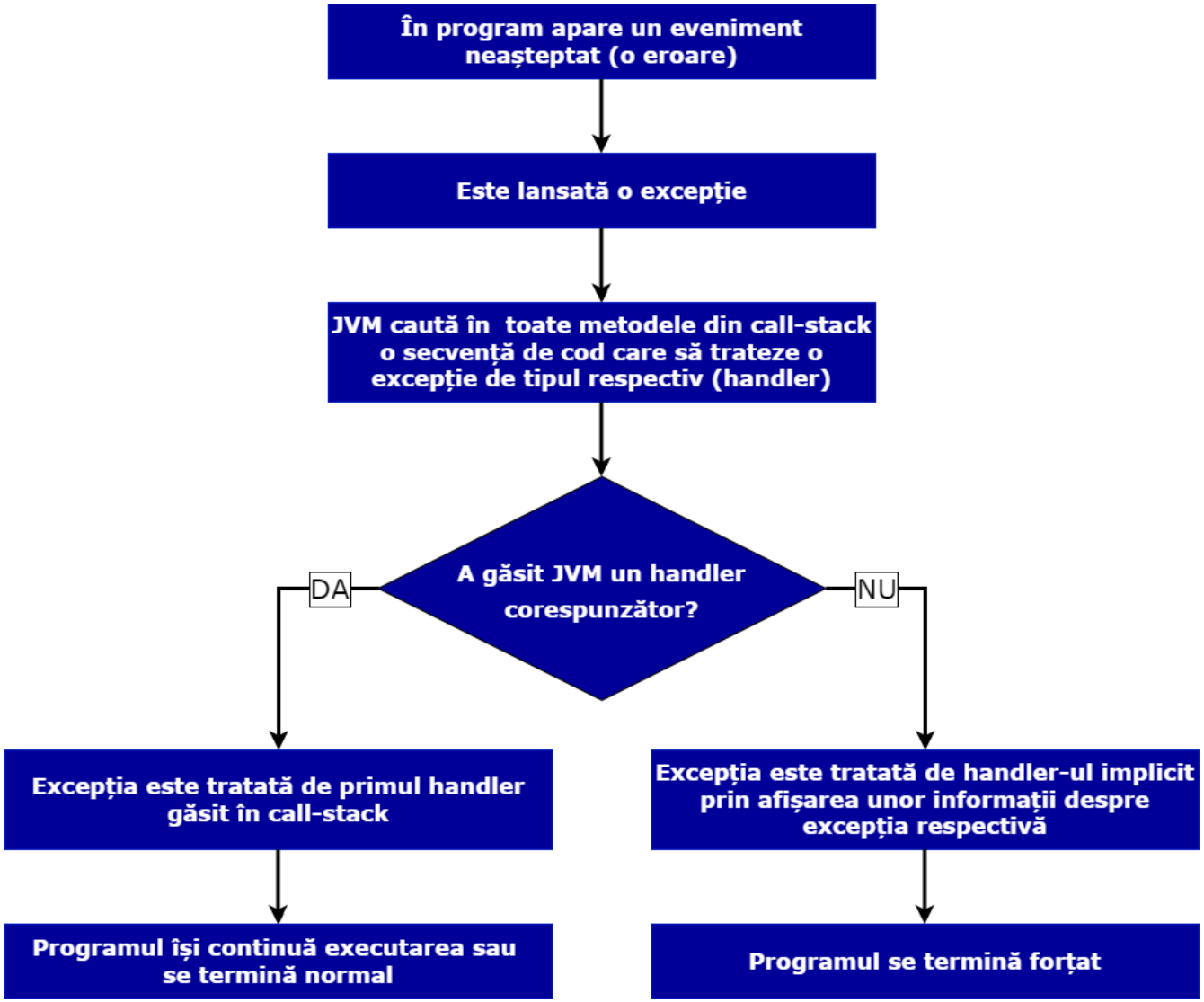
- **Generarea excepției**: JVM instanțiază un obiect al clasei `Exception` care încapsulează informații despre excepția apărută

- **Lansarea/aruncarea excepției:** obiectul generat este transmis mașinii virtuale

- **Propagarea excepției:** JVM parcurge în sens invers call-stack-ul, căutând un un cod care tratează acel tip de eroare, handler;

- **Prinderea și tratarea excepției**:
  - primul handler găsit în call-stack este executat ca reacție la apariția erorii
  - dacă nu se găsește niciun handler, atunci JVM oprește executarea programului și afișează un mesaj descriptiv de eroare

Mecanismul Java pentru manipularea excepțiilor



Instrucțiunea try - catch

```
try {
    bloc de instrucțiuni care poate produce excepții
}
catch(Excepție_A e) {
    Tratare excepție A
}
catch(Excepție_B e) {
    Tratare excepție B (mai generală)
}
finally {
    Bloc care se execută întotdeauna
}
```

Instrucțiunea try - catch

Exemplu	Cazuri

`try`
`{`
`  cod1;`
`  cod2;`
`  cod3;`
`}`
`catch(ExceptionClass ob)`
`{`
`  cod4;`
`}`
`cod5;`
	Cazul 1
	▪ nu apare nicio excepție în blocul `try`
	▪ se executa `cod1`, `cod2`, `cod3` și `cod5`
	
	Cazul 2
	▪ presupunem că apare o excepție în `cod2`
	▪ se execută `cod1`
	▪ punctul de executare se mută în blocul `catch`
	▪ dacă excepția este de tipul precizat in blocul `catch` se execută `cod4`
	▪ se execută `cod5`
	▪ nu se mai execută `cod3`

Instrucțiunea try - catch

-

Exemplu	Cazuri

`try`
`{`
`  cod1;`
`  cod2;`
`  cod3;`
`}`
`catch(ExceptionClass ob)`
`{`
`  cod4;`
`}`
`cod5;`
	Cazul 3
	▪ presupuem că apare o excepție în `cod2`
	▪ se execută `cod1`
	▪ punctul de executare se mută în blocul `catch`
	▪ dacă excepția nu este de tipul precizat in blocul `catch` programul își termină executarea cu o eroare!!!!

Observații

- Un bloc `try` poate arunca mai multe excepții care pot fi de tip diferit.
- Fiecare bloc `catch` poate intercepta excepții de tipul precizat în antetul său.
- La interceptarea mai multor excepții, ordinea blocurilor `catch` este importantă:
  - la aruncarea unei excepții într-un bloc `try`, blocurile `catch` sunt examinate în ordinea apariției
  - este executat primul bloc care se potrivește cu tipul de excepție

Observații

➢ Exemplu:

```
catch (Exception e)
{ . . . }
catch (NegativeNumberException e)
{ . . . }
```

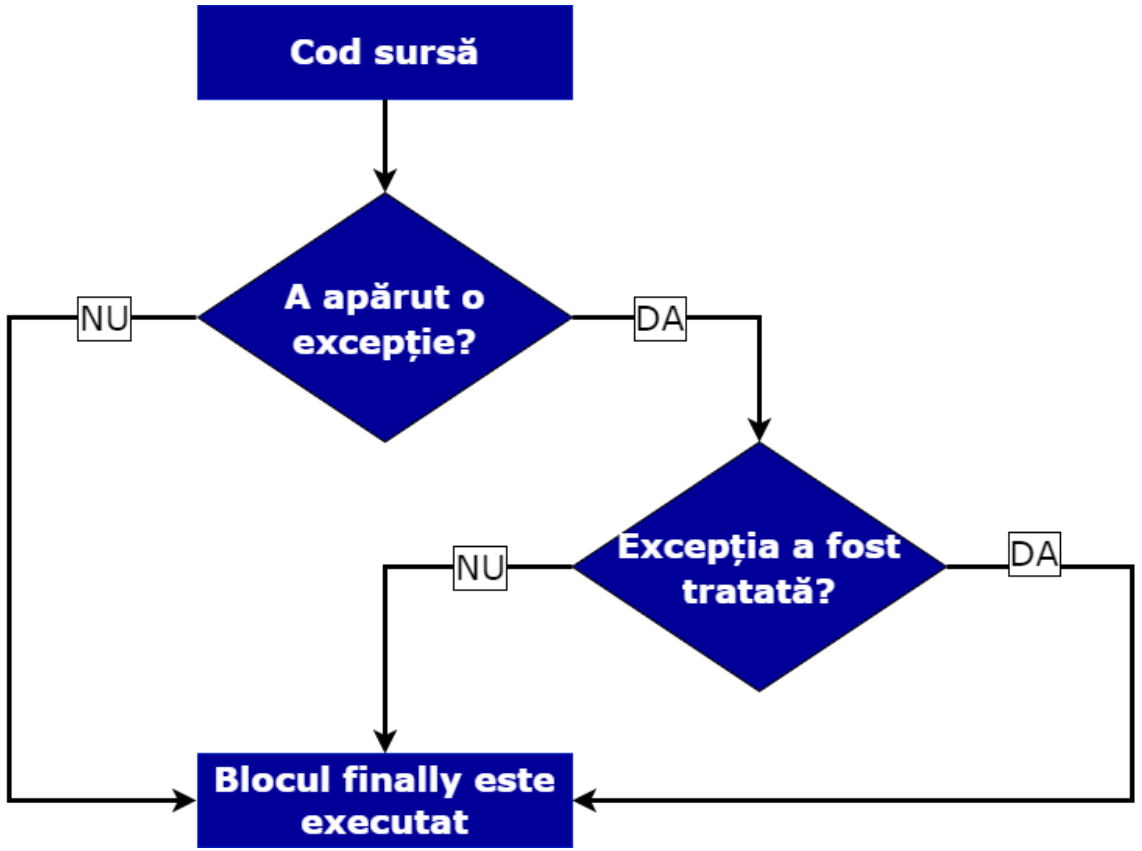
- Deoarece **NegativeNumberException** este un tip de **Exception**, toate **NegativeNumberException** vor fi interceptate de către primul bloc `catch` înainte de a se ajunge la cel de-al doilea.

- Blocul **catch** pentru **NegativeNumberException** nu se va executa!

- **Tipuri mai specifice de excepţii trebuie să apară la început, urmate de tipurile mai generale!!!**

Clauza finally

- Blocul `finally` nu are parametri și poate să lipsească, dar, dacă există, atunci se execută întotdeauna, indiferent dacă a apărut o excepție sau nu.



Clauza finally

```
Scanner fin;
try {
 fin = new Scanner(new File("numere.in"));
 int x = fin.nextInt();
 System.out.println(x);

 fin.close();
} catch (FileNotFoundException e) {
 System.out.println("Fisierul nu exista!");
}
catch(InputMismatchException e)
{
 System.out.println("Format gresit!");
}}
```

numere.in

12c

```
fin.close();
```

Nu se execută!!

Clauza finally

```
Scanner fin;
try {
 fin = new Scanner(new File("numere.in"));
 int x = fin.nextInt();
 System.out.println(x);
} catch (FileNotFoundException e) {
 System.out.println("Fisierul nu exista!");
}
catch(InputMismatchException e)
{
 System.out.println("Format gresit!");
}finally
{
    fin.close();
}
```

numere.in

12c

```
fin.close();
```

Se execută!!

Aruncarea unei excepții

- Dacă în corpul unei metode nu se tratează o anumită excepție sau un set de excepții, în antetul metodei se poate folosi clauza **throws** pentru ca acesta/acestea să fie tratate de către metoda apelantă.

- **Sintaxa:**

```
tip_returnat numeMetoda(<listă argumente>) throws listaExcepții
```

- **Exemplu**

```
void citire() throws IOException {
    System.in.read();
}
void citesteLinie(){
  citire();
}
```

Excepții definite de către programator

- Sunt situații în care trebuie să fie tratate excepții specifice, precum excepția dată de adăugarea unui element într-o stivă plină, introducerea unui CNP invalid, utilizarea unei date calendaristice anterioare unui proces etc.

- Se poate modela o anumită excepție printr-o clasă care extinde fie clasa **`Exception`**, fie clasa **`RuntimeException.`**

- Lansarea unei excepții se realizează prin clauza următoare:

```
throw new ExcepțieNouă(<listă argumente>)
```

Excepții definite de către programator

- Toate clasele predefinite pentru manipularea excepțiilor au următoarele proprietăți:
  - încapsulează un constructor cu un singur argument de tipul **`String`**
  - clasa are o metodă de acces, **`getMessage()`**, care poate accesează șirul dat ca argument constructorului la crearea obiectului excepție

- **Exemplu:** Implementarea unei stive de numere întregi folosind un tablou unidimensional, precum și excepții specifice

- Definim o clasă **`StackException`** pentru manipularea excepțiilor specifice unei stive:

```
public class StackException extends Exception {
   public StackException(String mesaj) {
            super(mesaj);
   }
}
```

Excepții definite de către programator

- Definim o interfață **`Stack`** în care precizăm operațiile specifice unei stive:

```
public interface Stack {
    void push(Object item) throws StackException;
    Object pop() throws StackException;
    Object peek() throws StackException;
    boolean isEmpty();
    boolean isFull();
    void print() throws StackException;
}
```

Excepții definite de către programator

- Definim o clasă **StackArray** în care implementăm operațiile definite în interfața **Stack:**

```
@Override
public void push(Object x) throws StackException {
     if(isFull())
          throw new StackException("Nu pot să adaug un element
                                    într-o stivă plină!");
     stiva[++varf] = x;
}
```

Excepții definite de către programator

- Definim o clasă **StackArray** în care implementăm operațiile definite în interfața **Stack**:

```
@Override
public Object pop() throws StackException {
     if(isEmpty())
          throw new StackException("Nu pot sa extrag un
                                   element dintr-o stivă vidă!");

     Object aux = stiva[varf];
     stiva[varf--] = null;
     return aux;
}
```

Excepții definite de către programator

➢Tratarea excepției:

```
public class Test_StackArray {
      public static void main(String[] args) {
            StackArray st = new StackArray(3);
            Random rnd = new Random();
            for(int i = 0; i < 20; i++)
            try {
                  int aux = rnd.nextInt();
                  if(aux % 2 == 0)
                        st.push(1 + rnd.nextInt(100));
                  else
                        st.pop();
                  st.print();
            }
            catch(StackException ex) {
                        System.out.println(ex.getMessage());
            }
      }
}
```

Tratarea unei excepții

- Începând cu Java 7, a fost introdusă instrucțiunea ***try-with-resources*** care permite închiderea automată a unei resurse, adică a unui surse de date de tip flux (de exemplu, un flux asociat unui fișier, o conexiune cu o bază de date etc.) .

Sintaxa:

```
try(deschidere Resursă_1; Resursă_2) {
    .........
}
catch(...) {
    .........
}
}
```

Tratarea unei excepții

➢ Exemplu

- Pentru a putea fi utilizată folosind o instrucțiune de tipul *try-with-resources*, clasa corespunzătoare unei resurse trebuie să implementeze interfața `AutoCloseable`.

- Toate tipurile de fluxuri bazate pe fișiere implementează interfața `AutoCloseable`, deci pot fi deschise utilizând o instrucțiune de tipul *try-with-resources*.

```
try(FileOutputStream fout = new FileOutputStream("numere.bin");
    DataOutputStream dout = new DataOutputStream(fout);)) {
    .........
  }
catch (...) {
    .........
}
```

Temtică curs 8

- Controlul moștenirii
- Mecanismul de tartare al excepțiilor
- Fluxuri

➢ În versiunea Java 17 a fost introdus **conceptul de clasă/interfață sealed** care permite un control detaliat al moștenirii prin precizarea explicită a subclaselor ce pot extinde o superclasă sau a claselor ce pot implementa o interfață.

- O clasă sealed se declară astfel:

```
[specificatori] sealed class Clasă permits
Subclase {
        ..........................
   }
```

- Dacă o clasă extinde o superclasă și/sau implementează anumite interfețe, atunci **cuvântul permits** și lista subclaselor care pot să extindă clasa respectivă se vor scrie la sfârșitul antetului său!

```
public sealed class Angajat extends Persoana
implements Comparable
    permits Economist, Paznic, Inginer {
    ..........................
}
```

- Declarările unei **clase sealed** și ale **subclaselor permise** trebuie să respecte următoarele reguli:

✓ clasa sealed și subclasele permise trebuie să facă parte din același modul sau, dacă sunt declarate într-un modul anonim, din același pachet;

✓ fiecare **subclasă permisă trebuie să extindă direct clasa sealed**;

✓ fiecare subclasă permisă trebuie să specifice în mod explicit modul în care va continua controlul moștenirii inițiat de superclasa sa, folosind exact unul dintre următorii modificatori:

- **`final`**: subclasa respectivă nu mai poate fi extinsă;
- **`sealed`**: subclasa respectivă poate fi extinsă doar în mod controlat (i.e., doar de subclasele pe care le permite explicit);
- **`non-sealed`:** subclasa respectivă poate fi extinsă fără nicio restricție (i.e., de orice altă clasă).

- Pentru clasa `Angajat` din exemplul se mai sus, o variantă de declarare a subclaselor poate fi următoarea:

```
public final class Paznic extends Angajat {
    ..........................
}
public non-sealed class Economist extends Angajat {
    ..........................
}
public sealed class Inginer extends Angajat
    permits InginerElectronist, InginerMecanic {
    ..........................  }
```

- În cazul unei interfețe, folosind modificatorul `sealed`, putem **specifica subinterfețele** care o pot extinde sau **clasele care o pot implementa**, astfel:

```
[public] sealed interface Interfață permits Subinterfețe, Clase {

        ..........................

    }
```

- O subinterfață care extinde o interfață sealed trebuie să respecte reguli asemănătoare celor precizate în cazul claselor sealed, cu observația că unei interfață îi putem aplica doar modificatorii `sealed` și `non-sealed`.

FLUXURI DE INTRARE/IEȘIRE

- Operațiile de intrare/ieșire sunt realizate, în general, cu ajutorul claselor din pachetul **java.io**, folosind conceptul de *flux* (stream).

- Un *flux* reprezintă o modalitate de transfer al unor informații în format binar de la o sursă către o destinație.

- În funcție de modalitatea de prelucrare a informației, precum și a direcției canalului de comunicație, fluxurile se pot clasifica astfel:

- *după direcția canalului de comunicație*:
  - ✓ de intrare
  - ✓ de ieșire
- *după modul de operare asupra datelor:*
  - ✓ la nivel de octet (flux pe 8 biți)
  - ✓ la nivel de caracter (flux pe 16 biți)

FLUXURI DE INTRARE/IEȘIRE

- *după modul în care acționează asupra datelor*:
  - ✓ primitive (doar operațiile de citire/scriere)
  - ✓ procesare (adaugă la cele primitive operații suplimentare: procesare la nivel de buffer, serializare etc

- În concluzie, pentru a deschide orice flux se instanțiază o clasă dedicată, care poate conține mai mulți constructori:
- un constructor cu un argument prin care se specifică calea fișierului sub forma unui șir de caractere;

- un constructor care primește ca argument un obiect de tip File;

› un constructor care primește ca argument un alt flux

FLUXURI DE INTRARE/IEȘIRE

- Clasa **`File`** permite operații specifice fișierelor și directoarelor, precum creare, ștergere, mutare etc., mai puțin operații de citire/scriere.

- **Metode uzuale ale clasei `File`:**

- `String getAbsolutePath()` – returnează calea absolută a unui fișier;

- `String getName()` – returnează numele unui fișier;

- `boolean createNewFile()` – creează un nou fișier, iar dacă fișierul există deja metoda returnează `false`;

› `File[] listFiles()` – returnează un tablou de obiecte `File` asociate fișierelor dintr-un director

FLUXURI DE INTRARE/IEȘIRE

➢ **Fluxurile primitive** permit doar operații de intrare/ieșire.

- După modul de operarea asupra datelor, fluxurile primitive se împart în două categorii:

1. **Prelucrare la nivel de caracter (fișiere text)**: informația este reprezentată prin caractere Unicode, aranjate pe linii (separatorul poate fi '`\r\n`' (Windows), '`\n`' (Unix/Linux) sau '`\r`' (Mac)).

- Informația fiind reprezentată prin caracter Unicode, se obține un flux pe 16 biți.
- **Deschiderea unui flux primitiv la nivel de caracter de intrare** se instanțiază clasa **`FileReader`**

```
FileReader fin = new FileReader("exemplu.txt");

File f = new File("exemplu.txt");
FileReader fin = new FileReader(f);
```

- Operația de citire a unui caracter se realizează prin metoda **`int read()`.**

FLUXURI DE INTRARE/IEȘIRE

- **Deschiderea unui flux primitiv la nivel de caracter** de ieșire se instanțiază clasa **`FileWriter`**

```
FileWriter fout = new FileWriter("exemplu.txt");
```

```
File f = new File("exemplu.txt");
FileWriter fout = new FileWriter (f);
```

- Operația de scriere a unui caracter se realizează prin metoda `void write(int ch).`

- Clasa `FileWriter` pune la dispoziție și alte metode pentru a scrie informația într-un fișier text:

- `public void write(String string)` - scrie în fișier șirul de caractere transmis ca parametru
- `public void write(char[] chars)` - scrie în fișier tabloul de caractere transmis ca parametru

➢ Observații

- Pentru deschiderea unui flux primitiv de ieșire la nivel de caracter în modul `append` (adăugare la sfârșitul fișierului), se utilizează constructorul:

  `FileWriter(String fileName, boolean append)`

- Deschiderea unui fișier impune tratarea excepției **`FileNotFoundException`**.

- Scrierea informației într-un fișier impune tratarea excepției `IOException`.

FLUXURI DE INTRARE/IEȘIRE

2. Prelucrare la nivel de octet(fișiere binare): informația este reprezentată sub forma unui șir octeți neformatați (2 octeți nu mai reprezintă un caracter) și nu mai există o semnificație specială pentru caracterele '\r' și '\n'.

- Deschiderea unui flux primitiv la nivel de caracter de intrare se instanțiază clasa **`FileInputStream`**

```
FileInputStream fin = new FileInputStream("exemplu.txt");

File f = new File("exemplu.txt");
FileInputStream fin = new FileInputStream(f);
```

- Operația de citire a unui caracter se realizează prin metoda **`int read()`**
- Citerea unui tablou de octeți

`int read(byte[] bytes)` //returnează numărul octeților citiți

FLUXURI DE INTRARE/IEȘIRE

- Deschiderea unui flux primitiv la nivel de caracter de ieșire se instanțiază clasa **`OutputFileStream`**

```
FileOutputStream fout = new FileOutputStream("test.txt");

File f = new File("exemplu.txt");
FileOutputStream fout = new FileOutputStream(f);
```

- Operația de citire a unui caracter se realizează prin metoda **`void write(int b).`**

- Scrierea unui tablou de octeți

`void write(byte[] bytes)//`returnează numărul octeților citiți

➢ **Exemplu: prelucrearea unei imagini BMP**

- Formatul BMP (bitmap) pe 24 de biți este un format de fișier binar folosit pentru a stoca imagini color bidimensionale având lățime, înălțime și rezoluție arbitrare.

- Fiecare pixel este codificat prin 3 octeți corespunzători intensităților celor 3 canale de culoare R (red), G(green) și B(blue).

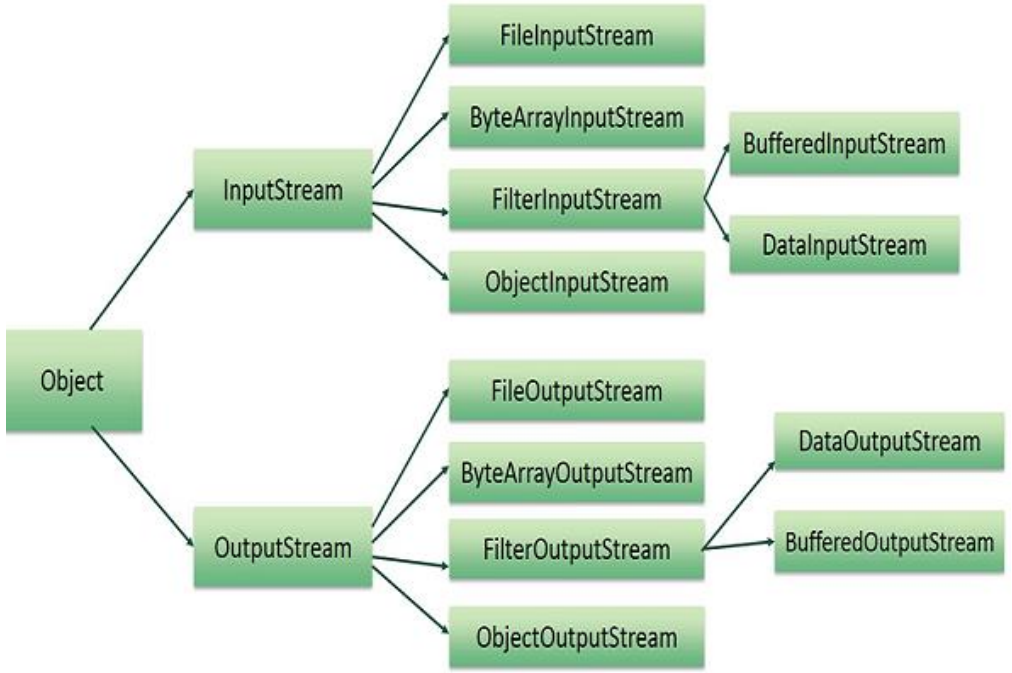
- Intensitatea fiecărui canal de culoare R, G sau B este dată de un număr natural cuprins între 0 și 255. De exemplu, un pixel cu valorile (0, 0 , 0) reprezintă un pixel de culoare neagră, iar un pixel cu valorile (255, 255, 255) unul de culoare albă.

- Formatul BMP cuprinde o zonă cu dimensiune fixă, numita header(54 octeți), și o zonă de date cu dimensiune variabilă care conține pixelii imaginii propriu-zise.

FLUXURI DE INTRARE/IEȘIRE

➢ Fluxuri de procesare

- Limbajul Java pune la dispoziție o serie de fluxuri de intrare/ieșire care au o structură stratificată pentru a adăuga funcționalități suplimentare pentru fluxurile primitive, întru-un mod dinamic și tr



FLUXURI DE INTRARE/IEȘIRE

➢ Fluxuri de procesare

- Constructorii claselor pentru fluxurile de procesare nu primesc ca argument un dispozitiv extern de memorare a datelor, ci o **referință a unui flux primitiv**.

```
FluxPrimitiv flux  = new FluxPrimitiv(<lista arg>);

FluxDeProcesare fluxProcesare = new FluxDeProcesare(flux);
```

FLUXURI DE INTRARE/IEȘIRE

- Fluxurile de procesare **`DataInputStream/DataOutputStream`**

- Fluxul procesat nu mai este interpretat la nivel de octet, ci octeții sunt grupați astfel încât aceștia să reprezinte date primitive sau șiruri de caractere (`String`).

DataInputStream	DataOutputStream
boolean readBoolean()	void writeBoolean(boolean v)
byte readByte()	void writeByte(byte v)
char readChar()	void writeChar(int v)
double readDouble()	void writeDouble(double v)
float readFloat()	void writeFloat(float v)
int readInt()	void writeInt(int v)
long readLong()	void writeLong(long v)
short readShort()	void writeShort(int v)
String readUTF()	void writeUTF(String str)

FLUXURI DE INTRARE/IEȘIRE

➢ Fluxuri de procesare pentru citirea/scrierea datelor folosind un buffer

- Fluxurile de procesare la nivel de buffer introduc în procesele de scriere/citire o zonă auxiliară de memorie, astfel încât informația să fie accesată în blocuri de caractere/octeți având o dimensiune predefinită.
- **Clase pentru citirea/scrierea cu buffer:**
  - `BufferedReader, BufferedWriter` – fluxuri de procesare la nivel de buffer de caractere
  - `BufferedInputStream, BufferedOutputStream` – fluxuri de procesare la nivel de buffer de octeți
- **Constructori:**
- `FluxProcesareBuffer flux = new FluxProcesareBuffer(new FluxPrimitiv("cale fișier"));`
- `FluxProcesareBuffer flux = new FluxProcesareBuffer(new FluxPrimitiv("cale fișier"), int dimBuffer);`
- Metodele uzuale ale acestor clase sunt: **`read/readline , write, flush`** (golește explicit buffer-ul, chiar dacă acesta nu este plin).

FLUXURI DE INTRARE/IEȘIRE

➢ Fluxuri de procesare de tip text pentru citirea/scrierea datelor formatate

- Clasa Scanner poate fi utilizată pentru citirea formatată a datelor de tip primitiv si a unui obiect de tip String, pentru un flux la nivel de caracter

- `Scanner flux = new Scanner(new File("cale fișier"));`

- Clasa conține metode dedicate pentru citirea formatată a datelor, precum și pentru parcurgerea fluxului:

✓ `Integer nextInt(), Double nextDouble(), String next(), String nextLine()`

✓ `boolean hasNext(), boolean nextDouble(), boolean next(), boolean nextLine()`

FLUXURI DE INTRARE/IEȘIRE

➢ Fluxuri de procesare de tip text pentru citirea/scrierea datelor formatate

- Clasa PrintWriter poate fi utilizată pentru scrierea formatată a datelor de tip primitiv si a unui obiect de tip String, pentru un flux la nivel de caracter

- `PrintWriter flux = new PrintWriter("cale fișier");`

- Clasa conține metode dedicate pentru scrierea formatată a datelor:

✓ `void write(TipDataPrimitiv data)`

✓ `void write(String ob)`