METODE AVANSATE DE PROGRAMARE

Conf.univ.dr. Ana Cristina DĂSCĂLESCU

Temtică curs 7

- Enumerare
- Mecanismul de tartare al excepțiilor

➢O **enumerare** este **un tip de data de referința** care poate încapsula o un set de constante.

- Sintaxa

```
public enum Denumire
{
    instante ale enumerarii
    [camp privat care retine valoarea unei constante]
    [constructor privat care instantiaza o referinta
enum]
    [o metoda care returneaza valoarea unei referinte]
}
```

- **Enumerare constante cu valori asociate**

```
 public enum Saptamana{
   LUNI(1), MARTI(2), MIERCURI(3), JOI(4), VINERI(5), SAMBATA(6),
DUMINICA(7);
     private final int zi;
     private Saptamana (int zi)
    {
        this.zi = zi;
    }
  public int getValue()
    {
       return zi;
    }
}
```

➢ Observații

- Orice enumerare este extinsă din clasa **`java.lang.Enum`** care conține o serie de metode specifice unui tip de data de referință enum, precum:

✓`String name`(): returnează numele unei instante a enumerării, stabilit la declararea sa

✓`int ordinal`(): returnează numărul de ordine al unei insatnțe a enumerării (prima instanță este indexată cu 0)

ENUMERĂRI

- ✓`String toString()`: returnează o reprezentare sub forma unui șir de caracter pentru o instanță a enumerării

- ✓Se poate obține o structură de date care să conțină toate valorile constantelor prin apelul metidei `values()`

›

```
for (Saptamana level : Saptamana.values()) {
    System.out.println(level);
}
```

- ✓O enumerare poate să încapsuleze metode statice

ENUMERĂRI

- ✓**O enumerare poate să încapsuleze o metodă abstractă**, în acest caz fiecare instanță enumerării trebuie să implementeze metoda abstractă (Enum Desing Pattern State Machine)

- ✓Pentru o instanță a unei enumerări se pot asocia mai multe valori!!!

CALL STACK

- **O excepție** este un **eveniment care întrerupe executarea normală a unui program**.

- Exemple de excepții: împărțirea unui număr întreg la 0, încercarea de deschidere a unui fișier inexistent, accesarea unui element inexistent într-un tablou, procesarea unor date de intrare incorecte etc.

- De regulă, rularea unui program presupune o succesiune de apeluri de metode!!!

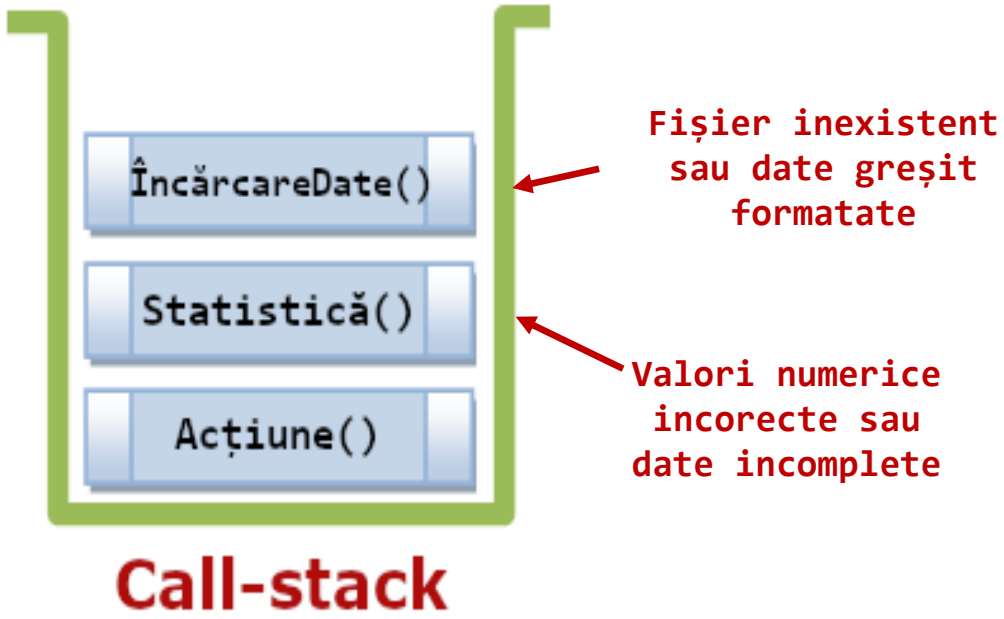
- Succesiunea de apeluri de metode a căror executare a început, dar nu s-a și încheiat este numită **call-stack (stiva cu apeluri de metode)** și reprezintă un concept important în logica tratării erorilor.

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



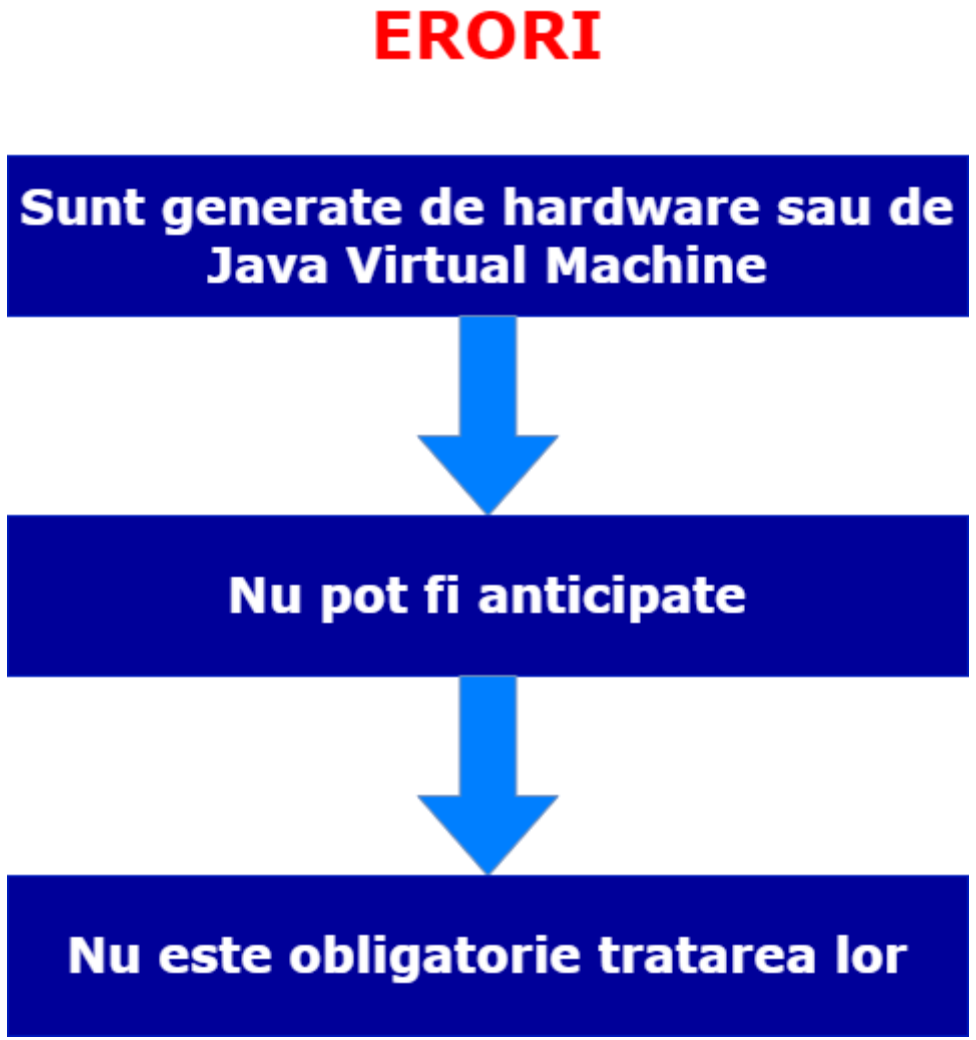
CALL STACK

- 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.

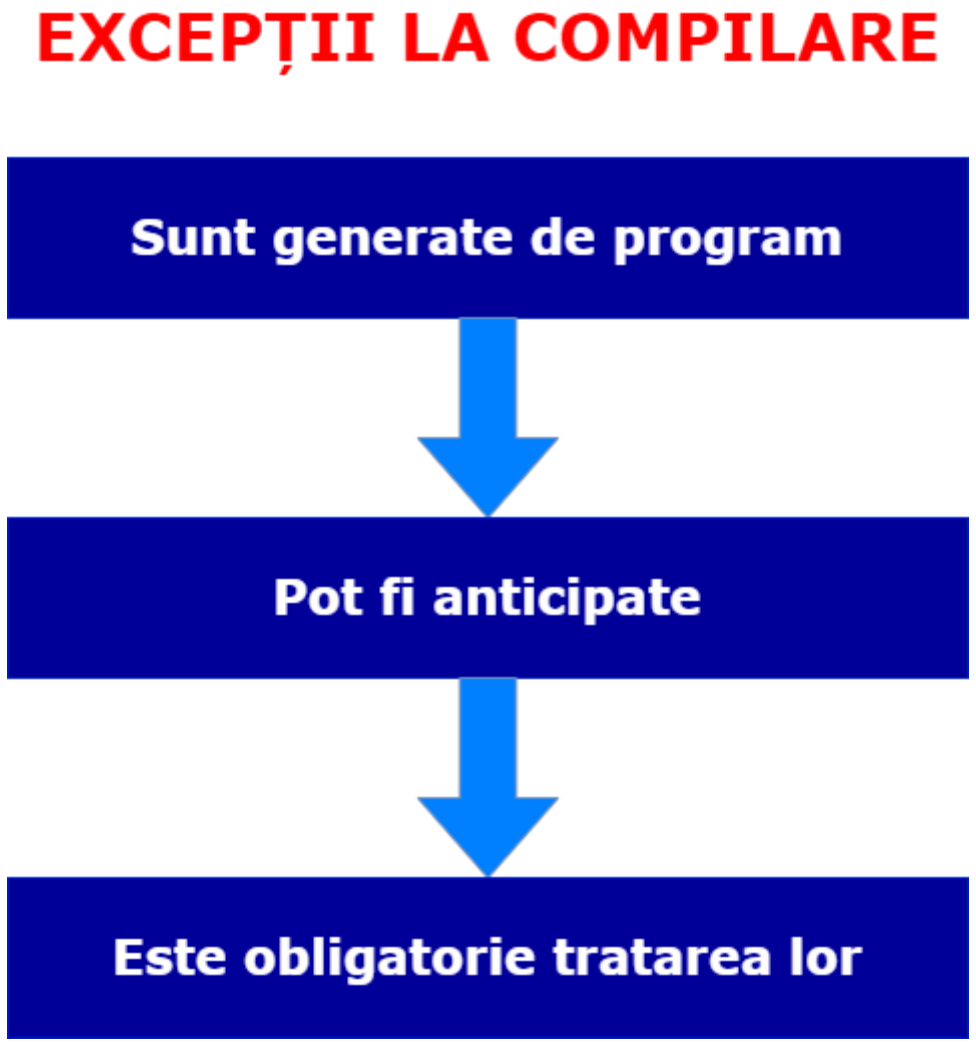
✓Practic, o excepție este un obiect care încapsulează detalii despre excepția respectivă, precum metoda în care a apărut, metodele din call-stack afectate, o descriere a sa etc.

Tipuri de excepții



Exemplu:
OutOfMemoryError

Tipuri de excepții



Exemplu:
IOException,
SQLException

Tipuri de excepții

EXCEPȚII LA RULARE

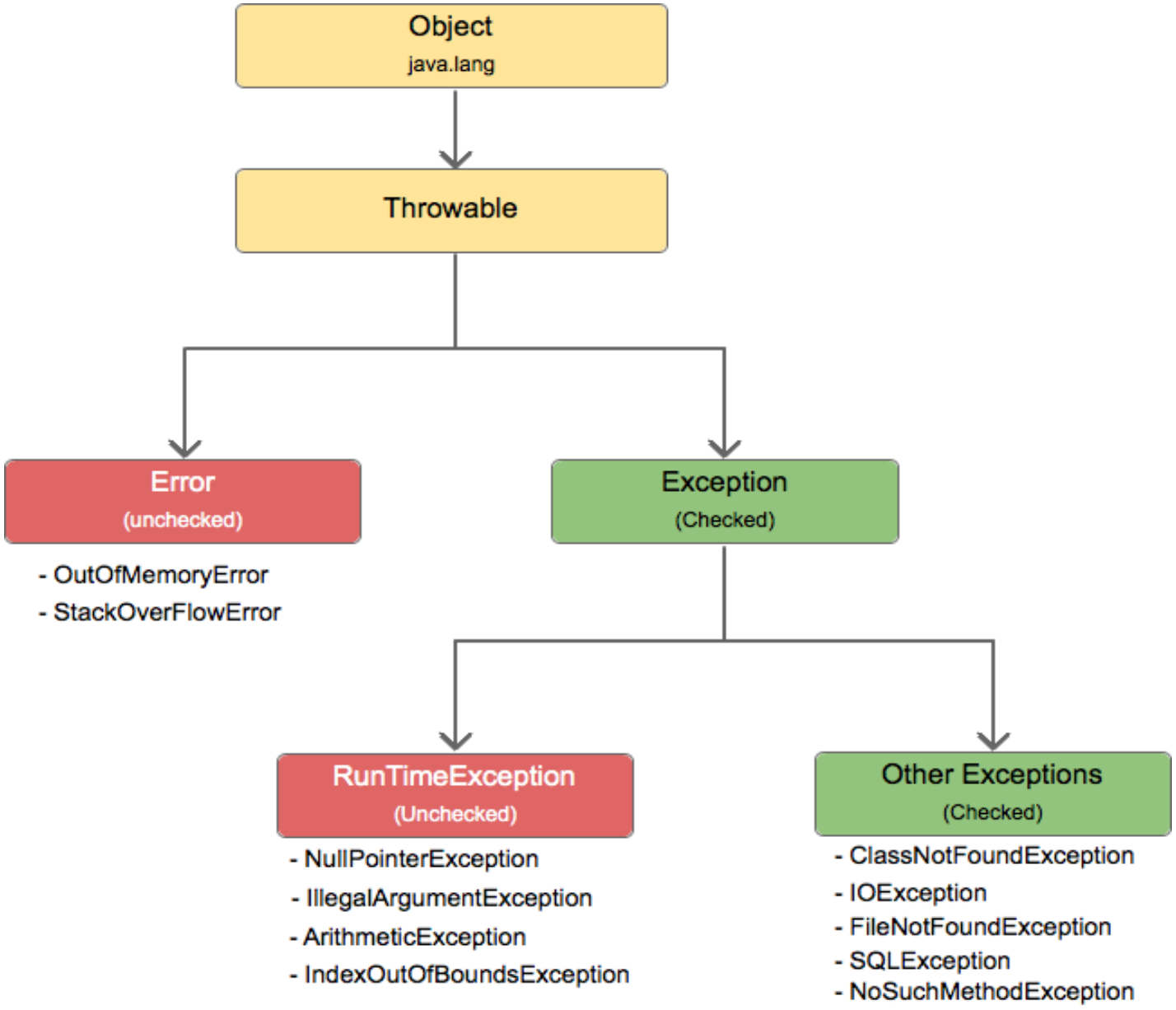
Sunt generate de o situație particulară care poate să apară în momentul executării programului

Pot fi foarte numeroase

Nu este obligatorie tratarea lor

Exemplu:
`IndexOutOfBoundsException`
`NullPointerException,`
`ArithmeticException`

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

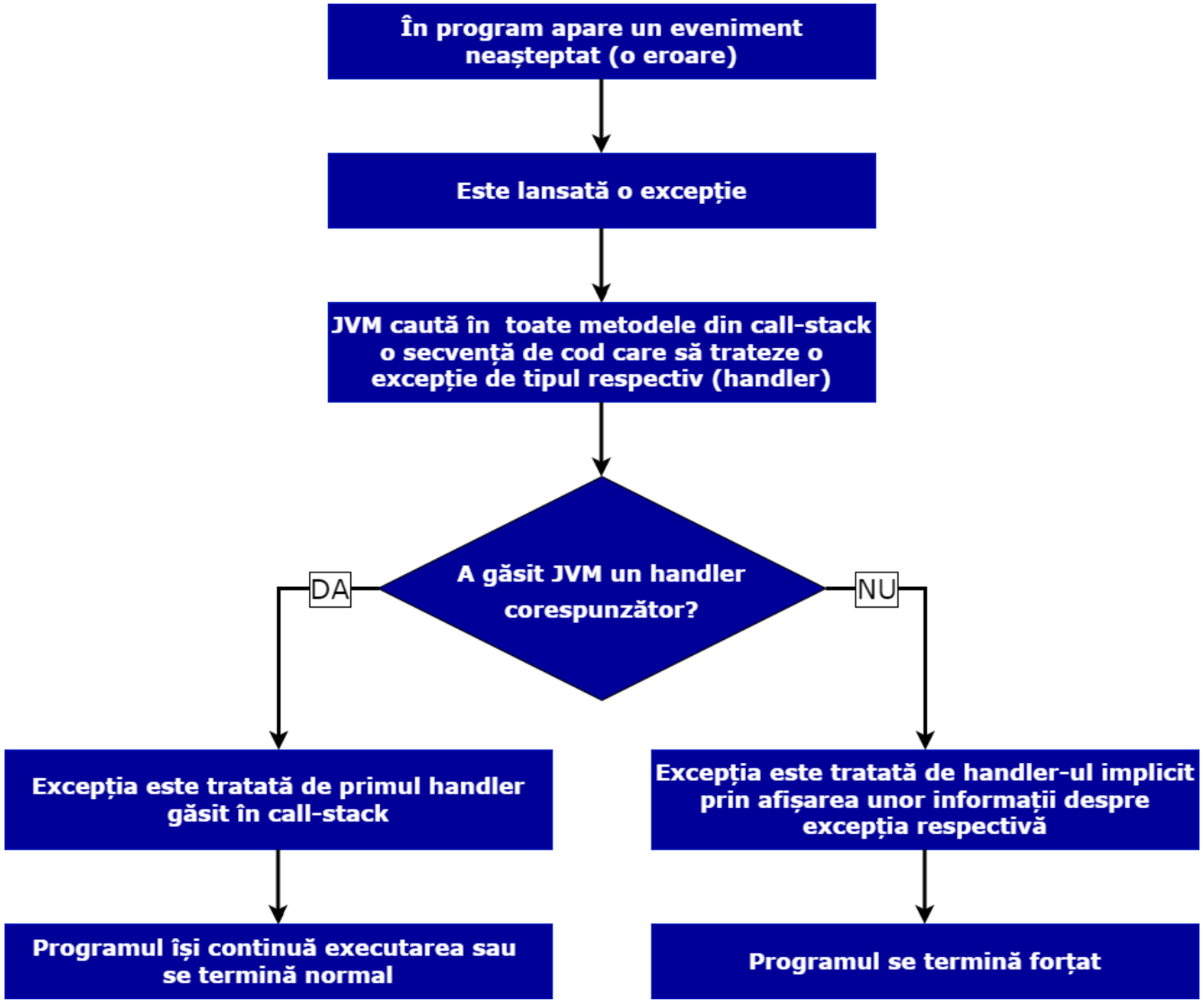
- **Generarea excepției**: JVM instanțiază un obiect al clasei `Exception` care încapsulează informații despre excepția apărută

- **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)
{ . . . }
```

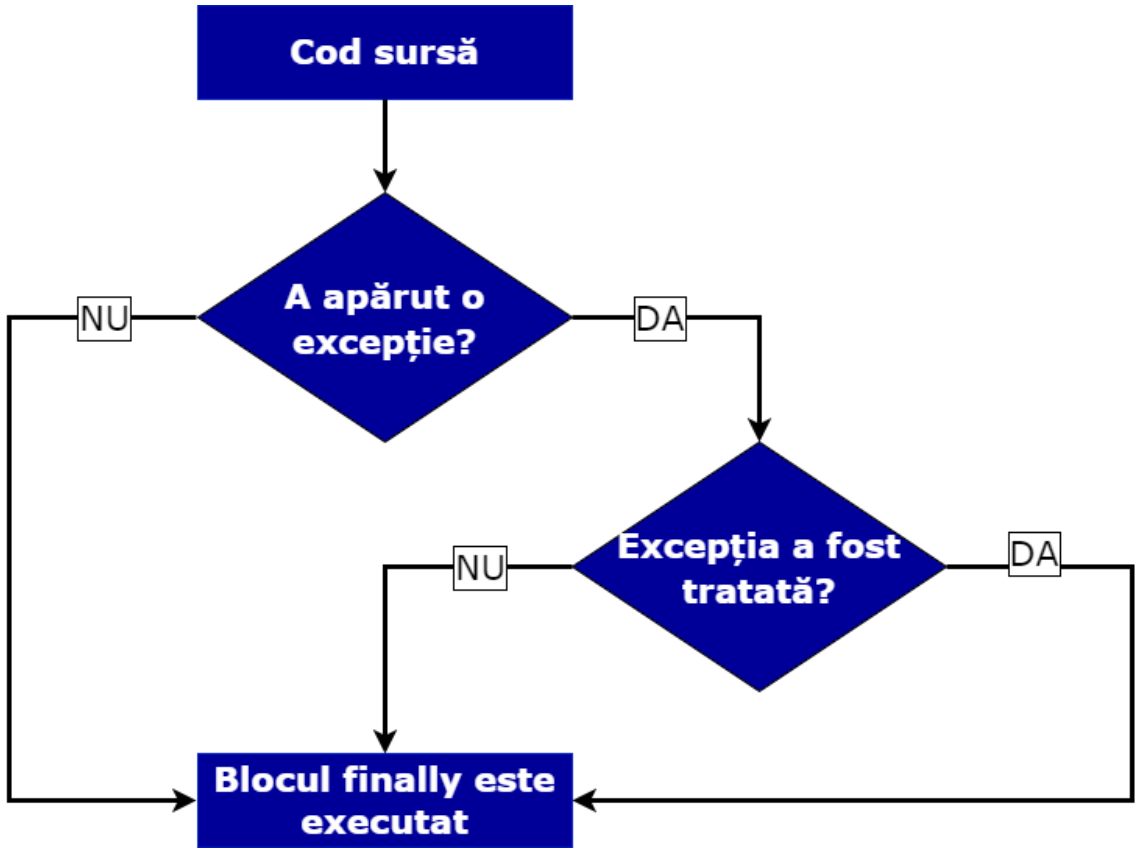
- Deoarece **NegativeNumberException** este un tip de **Exception**, toate **NegativeNumberException** vor fi interceptate de către primul bloc `catch` înainte de a se ajunge la cel de-al doilea.

- 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

➢ **Definirea unei excepții**

1. Definirea clasei pentru excepția personalizată

✓ o nouă clasă care să extindă clasa **`Exception/ RuntimeException`**

```
public class NumeExceptie extends Exception {
    // Constructori și alte metode pot fi definite aici
}
```

Excepții definite de către programator

2. Definirea constructorilor

```
public NumeExceptie() {
    super("Mesajul de eroare implicit");
}


public NumeExceptie(String mesaj) {
    super(mesaj);
}
```

Excepții definite de către programator

3. Tratarea excepției

✓utilizarea clasei de excepție personalizată în blocuri try-catch pentru a gestiona situațiile specifice

```
try {
    // Cod care poate arunca excepția personalizată
    throw new NumeExceptie("A apărut o eroare specifică");
} catch (NumeExceptie e) {
    // Tratarea excepției personalizate
    System.out.println("A fost prinsă o excepție personalizată: " +
e.getMessage());
}
```

Excepții definite de către programator

➢ **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,
- **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 (...) {
    .........
}
```