ELEMENTE AVANSATE DE PROGRAMARE

Conf.univ.dr. Ana Cristina DĂSCĂLESCU

Erori și Excepții

➢**Excepţie:** un eveniment care se produce în timpul execuţiei unui program, provocând întreruperea cursului normal al execuţiei acestuia.

➢**Tipuri de excepții:**

1. Erori: cauzate de echipamentul hardware sau erori JVM.

Exemple de erori: OutOfMemoryError, StackOverflowError etc.

- acestea nu sunt generate de codul sursă Java
- nu pot fi anticipate
- nu este obligatorie tratarea lor

Erori şi Excepţii

2. Excepții la complilare: sunt generate de către codul sursă

➢Exemple:

- deschiderea unui fişier inexistent: FileNotFoundException
- indexare eronată a unui element dintr-un tablou: ArrayIndexOutOfBoundsException
- formate de intrare necorespunzătoare: IOException
- erori apărute la interogarea serverelor de baze de date SQLException

➢Aceste excepții pot fi anticipate

➢Este obligatorie tratarea lor!!!!

Erori și Excepții

3. Excepții la executare: pot fi generate de o situație particulară

➢Exemple:

- folosirea unei referințe cu valoarea null pentru accesarea unui membru de obiect NullPointerException
- operațiuni aritmetice ilegale (ex: împărțire la 0) ArithmeticException
- argument incorect într-o metodă IlegalArgumentException

➢Aceste excepții pot fi anticipate

➢Pot fi tratate:

- prin furnizarea unei excepții, care poate intrerupe executarea programului
- prin cod

Excepţii la compilare- exemplu

```
import java.util.Scanner;
public class Exemplu{
    public static void main(String[] args){
        Scanner tas = new Scanner(System.in);
        System.out.print("Dati valoarea: ");
        int x = tas.nextInt();
    System.out.println("Valoare: " + (x + 1)); }}
```

Dati valoarea: sir

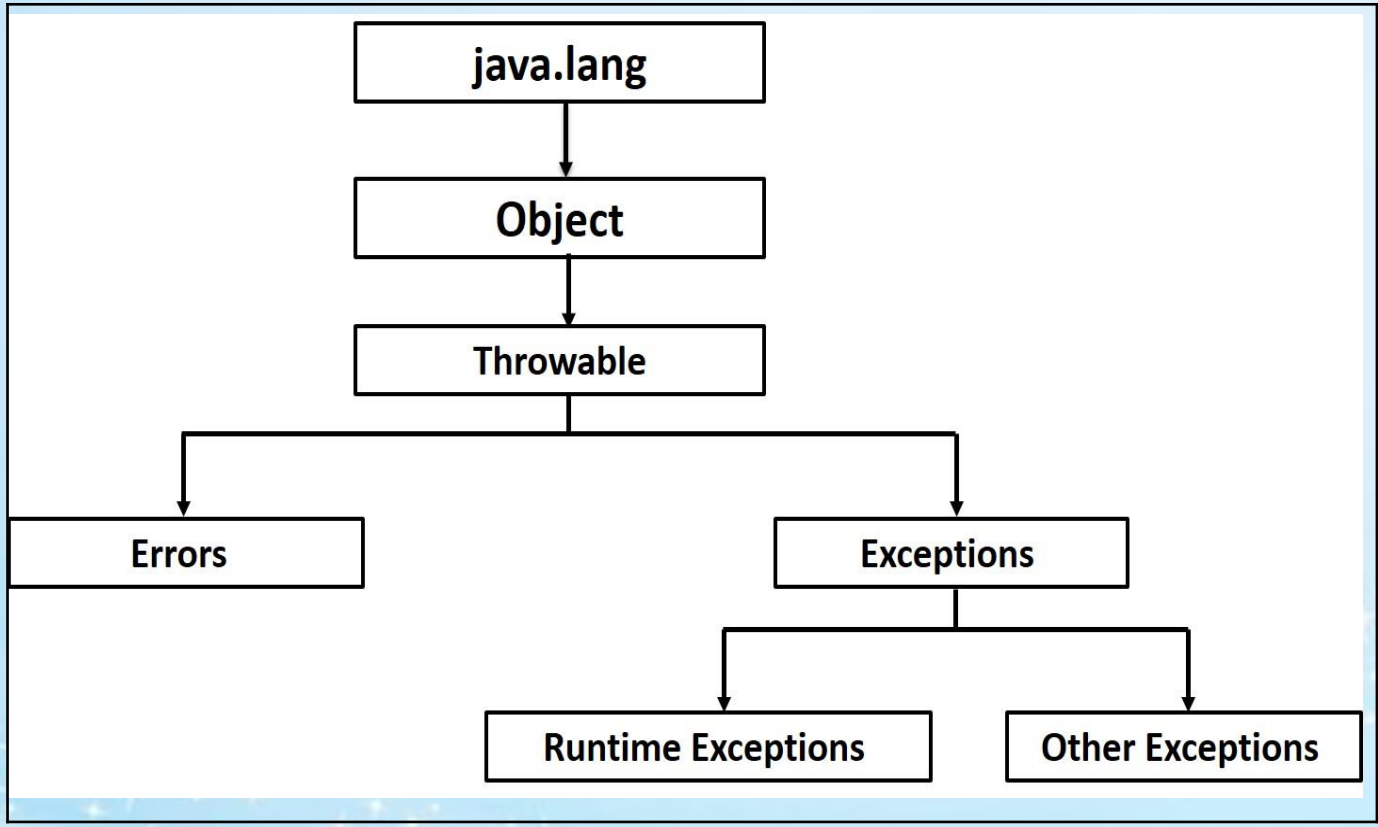
```
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:909)
        at java.util.Scanner.nextInt(Scanner.java:2119)
        at Exceptii.Exemplu.main(Test_Exceptie.java:16)
Java Result: 1
```

Sistemul de excepţii: principii şi avantaje

- eroarea nu mai este o simplă valoare particulară, ci devine un obiect, putând astfel încapsula orice fel de informaţie necesară despre eroarea apărută;
- gruparea erorilor după tipul lor (de exp. IOException -> FileNotFoundException, EOFException)
- propagarea excepţiei în sens invers prin call stack este automatizată de către mașina virtuală.
- excepțiile pot fi tratate și în alt loc decât cel în care s-au generat.

Erori și Excepții

- O eroare este reprezentată de un obiect de clasa Error
- O excepție este reprezentată de un obiect de clasa Exception



Metode relevante din clasa *Throwable:*

Throwable(String) - stabilește mesajul încapsulat.

String getMessage() - extrage mesajul încapsulat in obiectul excepție.

getStackTrace() și printStackTrace() - permit accesul la elementele call stack-ului.

Mecanismul folosit pentru manipularea excepțiilor

- JVM creează un obiect din clasa Exception care încapsulează informaţii despre excepţia respectivă.

  Numim acest pas *generarea excepţiei*

- Obiectul este înmânat maşinii virtuale

  Numim acest pas *aruncarea excepţiei*

- Maşina virtuală parcurge în sens invers call stack-ul, căutând un handler (o porțiune de cod care tratează acel tip de eroare).

  Numim acest pas *propagarea excepției*

- Primul handler găsit într-una dintre metodele din call stack este executat ca reacție la apariția erorii.

  Numim acest pas *prinderea şi tratarea excepţiei*

"Prinderea" şi tratarea excepţiilor

➢Tratarea exceptiilor se realizează prin intermediul blocurilor de instrucțiuni try-catch-finally

```
try {
      // Instrucțiuni care pot genera excepții      }
      catch (TipExceptie1 obiect) {
      // Tratarea excepțiilor de tipul 1}
      catch (TipExceptie2 obiect) {
      // Tratarea excepțiilor de tipul 2}
      . . .
      finally {
      // Cod care se execută indiferent dacă apar sau nu  excepții }
```

“Prinderea” şi tratarea excepţiilor

➢Evoluția execuţiei este următoarea:

- dacă în blocul try nu apare nicio excepţie, atunci instrucţiunile acestuia se vor executa până la final şi apoi se continuă cu instrucţiunile ce urmează construcţiei try...catch

- dacă în blocul try apare o excepţie, acesta se încheie prematur. Maşina virtuala preia obiectul excepţie şi caută un handler pentru el.

- dacă există un bloc catch ataşat care prinde fie acel tip exact de excepție, fie unul părinte, instrucțiunile acelui bloc catch vor fi rulate pe post de handler.

- dacă nu există un bloc catch care prinde excepţia respectivă, maşina virtuală va încheia prematur execuția metodei şi va începe să caute un handler în metodele anterioare din call stack.

“Prinderea” şi tratarea excepţiilor

Observaţii:

- Se pot grupa mai multe instrucţiuni generatoare de excepţii în blocul **try.**
- Dacă între tipurile de excepţii generate nu există nicio relaţie de moștenire, se poate folosi câte un bloc catch pentru fiecare tip de excepție, în orice ordine.
- Dacă unele tipuri de excepții sunt derivate din altele, ordinea blocurilor **catch** trebuie să fie de la particular către general (prima dată vor fi tratate tipurile derivate şi abia apoi tipurile părinte).
- Începând cu Java SE 7 există posibilitatea ca un singur bloc **catch** să prindă/trateze mai multe tipuri de excepții.

Blocul finally

- Oferă garanţia executării de cod la finalul constructiei **try...catch,** indiferent daca a apărut sau nu o excepţie.
- În general este utilizat pentru eliberarea de resurse (fişiere sau conexiuni de rețea deschise).
- Blocul finally va fi executat întotdeauna după blocurile try şi catch, după cum urmează:
  - dacă în try nu apare o excepţie, finally este executat imediat după *try*
  - dacă în try este aruncată o excepţie:
    - ✓ dacă există un bloc catch corespunzător, acesta va fi rulat după întreruperea execuţiei lui try, urmat de finally
    - ✓ dacă nu există bloc catch, se execută finally imediat după try şi abia apoi se părăseşte metoda curentă, căutând un handler în cea anterioară din call stack.

“Prinderea” şi tratarea excepţiilor

- Blocurile **catch** se exclud reciproc: nu exista nicio situație în care “se aruncă mai multe excepții”.

Exemplu: scrierea valorii 10 într-un fişier

```
try{
        FileOutputStream fos = new FileOutputStream(new File("test.txt"));
        fos.write(10);
        fos.close();
        } catch (NotFoundException ex) {
System.out.println("Fișierul dorit nu există");
} catch (IOException ex) {
System.out.println("Nu se poate scrie în fișier");
```

Nu se poate scrie o valoare

Fişierul poate să nu existe

Conversie greşită

Aruncarea excepțiilor

- O metodă în care se generează o excepție poate sa o trateze, sau să o "arunce" către alte metode care o apelează.
- Se folosește clauza throws

```
[modificatori] TipReturnat metoda([argumente])
throws TipExceptie1, TipExceptie2, ...
{
    ...
}
```

- O metodă care apelează o metode ce a "aruncat" o excepție, fie ignoră excepția respectivă (adică o "aruncă" mai departe), fie să o tratează.

Excepţii definite de către programator

- Un cod sursă Java poate sa trateze erori care nu au fost prevăzute în ierarhia standard.
- Unele excepţii standard nu descriu clar o eroare.
- Programatorul poate să definească o excepţie proprie, însă aceasta trebuie să se încadreze în ierarhia excepțiilor Java;

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

// Apelează constructorul superclasei Exception
}
}
```

Fluxuri standard de intrare și ieșire

- În orice program Java există:
  - o intrare standard;
  - o ieşire standard;
  - o ieşire standard pentru erori.

- Intrarea/ieşirea standard sunt reprezentate de obiecte ce descriu fluxuri de date care comunică cu dispozitivele standard (tastatură, monitorul).
- Obiectele sunt definite în clasa System:
  - **System.in** = flux de intrare de tip InputStream
  - **System.out** = flux de ieșire de tip PrintStream
  - **System.err** = flux pentru eroare de tip PrintStream

Fluxuri standard de intrare și ieșire

➢Citirea datelor de la tastatură

1. Se foloseşte un obiect al clasei **BufferedReader** din pachetul java.io

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

- readline() – citeşte o linie de la tastatură;

Se impune tratarea unei excepții de intrare/ieșire.

2. Se foloseşte un obiect de tip **Scanner** din pachetul java.util

```
Scanner in= new Scanner(System.in);
```

- next(), nextInt(), nextLine() etc.

Fluxuri

- Un **flux** este un canal de comunicaţie serial unidirecțional între două procese pe 8 biţi sau 16 biţi.

- Indiferent de tipul informaţiilor, citirea/scrierea de pe/către un mediu extern se respectă următorul algoritm:

```
deschide canal comunicație
while (mai sunt informații) {
citește/scrie informație;}
închide canal comunicație;
```

- Clasele și intefețele standard pentru lucrul cu fluxuri se găsesc în pachetul **java.io.**

Clasificarea fluxurilor

➢**După direcţia canalului de comunicaţie** deschis fluxurile se împart în:

- fluxuri de intrare: fluxuri de date prin intermediul cărora aplicaţia citeşte date din surse externe
- fluxuri de ieşire: fluxuri pentru trimiterea informaţiei din aplicaţie către destinaţii externe (fişiere, alte maşini virtuale aflate în reţea etc.)

➢**După tipul de date pe care operează**:

- fluxuri de octeţi (comunicarea serială se realizează pe 8 biţi)
- fluxuri de caractere (comunicarea serială se realizează pe 16 biţi)

➢**După acţiunea lor**:

- fluxuri primare de citire/scriere a datelor
- fluxuri pentru procesarea datelor

Fluxuri primitive

- Sunt responsabile cu citirea/scrierea efectivă a datelor, punând la dispoziţie implementări ale metodelor de bază **read/write**.

În funcție de tipul sursei datelor, ele pot fi împărțite astfel:

`FileReader`
`FileWriter` — *pentru citire/scriere la nivel de caracter*

`FileInputStream`
`FileOutputStream` — *pentru citire /scriere la nivel de octet*

Crearea unui flux

➢ **FluxPrimitiv numeFlux =new FluxPrimitiv (dispozitivExtern);**

- crearea unui flux de intrare pe caractere

```
FileReader in = new FileReader("fisier.txt");
```

- crearea unui flux de ieşire pe caractere

```
FileWriter out = new FileWriter("fisier.txt");
```

- crearea unui flux de intrare pe octeţi

```
FileInputStream in=newFileInputStream("fisier.dat");
```

- crearea unui flux de ieșire pe octeţi

```
FileOutputStrem out=new
    FileOutputStream("fisier.dat");
```

Citirea/scrierea formatată

➢Clasa Scanner

- disponibilă în pachetul java.util şi permite citirea datelor de tip primitiv sau de tip şir, folosind expresii regulate.

```
Scanner sc = new Scanner(new File("fisier"));
```

- conţine metode pentru citire: nextInt(), nextln() etc.

➢Clasa FileWriter

- disponibilă în pachetul java.util şi permite afișarea datelor de tip primitiv şi de tip şir.

```
PrintWriter fout=new PrintWriter("fis.txt");
```