



Programare orientată pe obiecte

1. Erori și excepții în Java

TECHNICAL UNIVERSITY

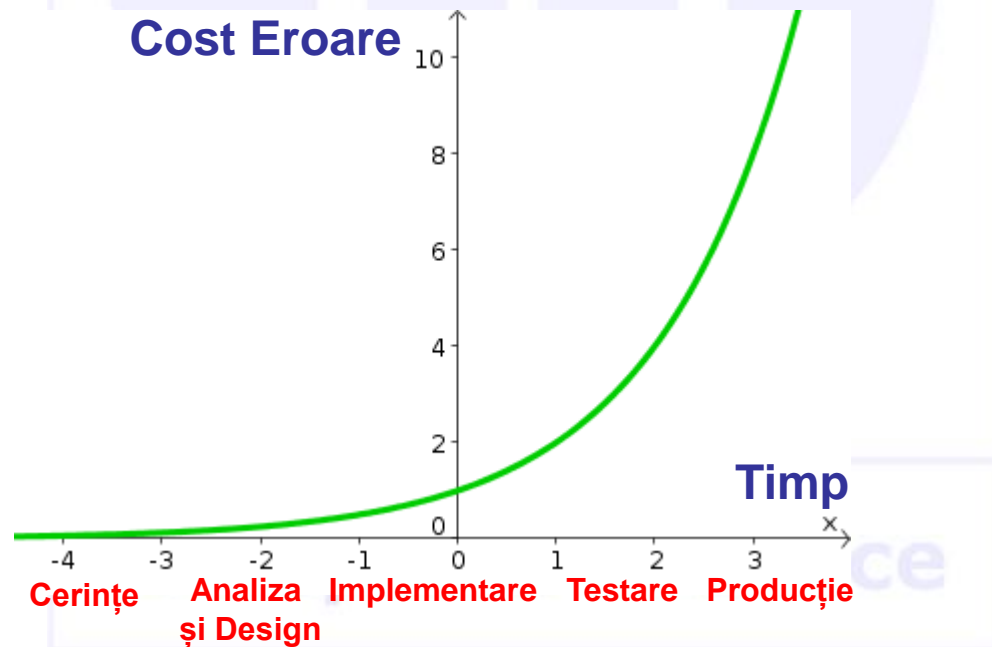
OF CLUJ-NAPOCA

Computer Science



De ce mecanisme de tratare a erorilor?

- Costul de remediere a unei erori nerezolvate poate crește exponențial odată cu timpul:





Soluții în Java

- Soluții
 - Aserțiuni
 - Mecanisme de tratare a excepțiilor
 - Mecanisme de testare – Testare Unitară

OF CLUJ-NAPOCA
Computer Science



Aserțiuni

- Aserțiunile sunt niște mecanisme ce permit testarea logicii programului în **faza de dezvoltare**
- În **faza de producție** aserțiunile se dezactivează, programul rulând normal fără a ține cont de existența aserțiunilor
- Avantajul principal al folosirii aserțiunilor
 - Timpul de implementare / testare scăzut – deoarece în unele cazuri nu merită efortul scrierii de cod pentru tratarea unor excepții care în faza de producție se știe că nu au cum să apară
- Raționament:
 - Presupunem (**assert**) totdeauna că o anumită condiție este adevărată, ex:
`assert (x>y)`
 - În cazul în care condiția nu este îndeplinită, programul va arunca o **AssertionError**



Aserțiuni

■ Sintaxa:

■ `assert expr1;`

`expr1` – expresie booleană,
valoarea “false” indică o eroare

sau

■ `assert expr1 : expr2`

`expr1` – expresie booleană
`expr2` – valoare, mesajul erorii



Activarea/dezactivarea aserțiunilor

- **Activare:**

```
java [ -enableassertions | -ea ] [ :<package name>"..." | :<class name> ]
```

- **Dezactivare:**

```
java [ -disableassertions | -da ] [ :<package name>"..." | :<class name> ]
```

- **Observație:** în alte medii de dezvoltare, opțiunea de utilizare a aserțiunilor trebuie activată/dezactivată explicit
 - În Eclipse opțiunile de mai sus sunt introduse prin accesarea Run Configurations -> tab-ul Arguments -> căsuța "VM Arguments", unde se inserează -ea sau -da



Aserțiuni: exemplu simplu

```
public static void main(String[] args) {  
  
    String result = null;  
  
    /*  
    ... Cod calcul rezultat  
    */  
  
    //Se presupune ca nu este legal sa avem valori null  
    assert result != null : "Nu sunt acceptate valori null";  
  
    System.out.println("end");  
  
}
```



Aserțiuni: când se folosesc

- Se folosesc aserțiuni doar pentru a verifica validitatea unor condiții care în mod normal nu ar avea cum să fie negative sub nici o formă!
- Nu folosiți aserțiuni de exemplu pentru a valida datele de intrare ale programului
 - Soluția oportună pentru o astfel de validare este folosirea mecanismelor de tratare a excepțiilor
 - În cazul în care datele de intrare nu sunt cele dorite, greșeala se poate trata cerând utilizatorului să reintroducă datele respective

Computer Science



Unde se pun aserțiunile

- **Locuri posibile**
 - **Precondiția** metodei – se verifică ce trebuie să fie adevărat atunci când se execută o metodă
 - **Postcondiția** metodei – ce trebuie să fie adevărat după ce s-a executat metoda
 - **Invarianti interni** – presupuneri că anumite porțiuni de cod sunt adevărate tot timpul
 - **Invariantul** clasei – ce trebuie să fie adevărat tot timpul legat de variabilele de instanță



Aserțiuni: alte exemple

```
public int calculeazaLungimeString(String inString)
{
    //PRECONDITIE
    assert inString != null : "Nu sunt acceptate valori null";

    int lungime = -1;

    lungime = inString.length();
    /*
     * ... cod ...
     */

    //POSTCONDITIE
    assert lungime >= 0 : lungime + " < 0";

    return lungime;
}
```



Aserțiuni: alte exemple

- Invarianti interni

```
if (i % 3 == 0) {  
    ...  
} else if (i % 3 == 1) {  
    ...  
} else {  
    assert i % 3 == 2 : i;  
    ...  
}
```

- Invarianti ai instrucțiunilor de control

```
void foo() {  
    for (...) {  
        if (...)  
            return;  
    }  
    assert false; /* Execuția nu ar trebui să ajungă în acest punct  
                  NICIODATA! */  
}
```



Aserțiuni: alte exemple

- Invariantul clasei – proprietățile clasei nu se schimbă niciodată atât înainte cât și după execuția oricărei metode
 - Exemplu: clasa ce implementează un arbore binar echilibrat; un invariant al clasei ar fi că **arborele este echilibrat tot timpul**
- Se poate introduce o metodă ce testează dacă un arbore este echilibrat sau nu

```
private boolean balanced() {  
    ...  
}
```

- Fiecare metodă sau constructor public al clasei va trebui să conțină această constrângere imediat înainte de return

```
assert balanced();  
return ...;
```



Excepții. Probleme în cursul execuției programelor

- Un program întâlnește adesea probleme (**excepții**) în cursul execuției sale:
 - poate avea probleme la citirea datelor,
 - pot exista caractere nepermise în date sau
 - indexul unui tablou poate depăși limitele acestuia
- Excepțiile Java permit programatorului să trateze astfel de probleme
 - Putem scrie programe care își revin la întâlnirea excepțiilor și își continuă execuția
 - **Programele nu trebuie să eșueze atunci când utilizatorul face o greșală!**
- În special intrarea și ieșirea sunt susceptibile la excepții
- Tratarea excepțiilor este esențială pentru programarea I/E



Exemplu de apariție a unei excepții

■ Programul:

```
import java.util.Scanner;
public class InputMismatchExceptionDemo {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        System.out.print("Enter one integer:");
        int inputNumber = keyboard.nextInt();
        System.out.println("The square of " + inputNumber + " is " + inputNumber *
            inputNumber);
    }
}
```

■ Cu intrarea: Enter one integer:h1

■ Are rezultatul:

```
java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:819)
    at java.util.Scanner.next(Scanner.java:1431)
    at java.util.Scanner.nextInt(Scanner.java:2040)
    at java.util.Scanner.nextInt(Scanner.java:2000)
    at InputMismatchExceptionDemo.main(InputMismatchExceptionDemo.java:11)
```



Discuție asupra exemplului

- Programul nu este greșit
 - Problema este că `nextInt` nu poate converti șirul de caractere "h1" la un `int`
 - În momentul în care `nextInt` a întâlnit problema, metoda a **aruncat** o excepție de tipul `InputMismatchException`
 - Sistemul de execuție Java a interceptat (a "prins") excepția, a oprit programul și a tipărit mesajele de eroare

OF CLUJ-NAPOCA
Computer Science



Excepții și erori

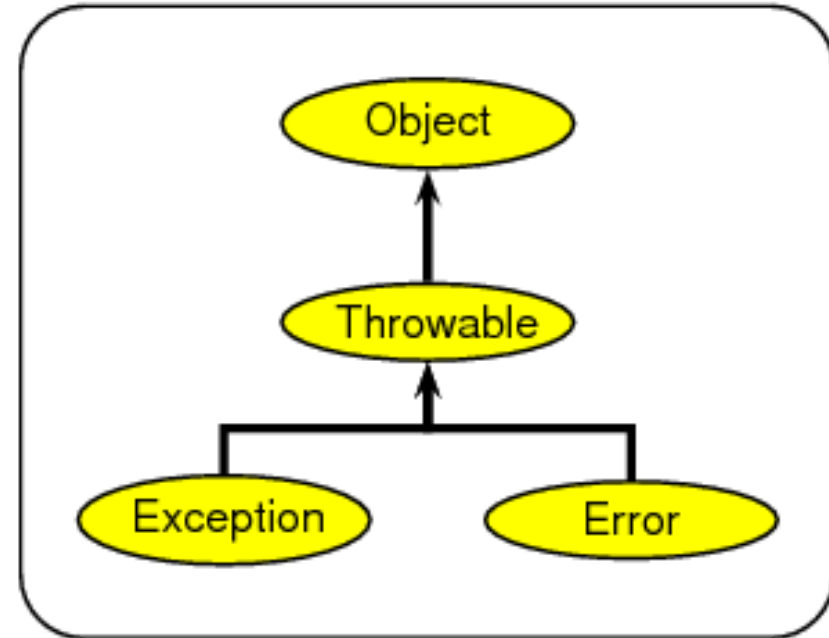
- O **excepție**: o problemă care apare în cursul execuției unui program
 - La apariția unei excepții, JVM creează un obiect de clasa **Exception** care conține informații despre problema apărută
 - Însuși programul Java poate **intercepta (catch)** o excepție. Apoi poate folosi obiectul de tipul excepție pentru a-și reveni după problemă
- Și o **eroare** este o problemă care apare la rularea unui program
 - O eroare este reprezentată de un obiect de clasa **Error**
 - Dar o eroare este prea severă pentru a fi tratată de un program. Programul trebuie să-și **înceteze execuția**

Computer Science



Ierarhia Throwable ("aruncabil")

- Atât clasa **Exception** cât și clasa **Error** extind clasa **Throwable**
 - O metodă Java poate "arunca" un obiect de clasa **Throwable**
 - D.e. `Integer.parseInt("zzz")` aruncă o excepție atunci când încearcă să convertească "zzz" într-un întreg
- **Excepții != Erori**: se pot scrie programele astfel încât să-și revină după excepții, dar nu se pot scrie astfel încât să-și revină după erori





Tratarea excepțiilor.

Mecanismul `try-throw-catch`

- Calea fundamentală pentru tratarea excepțiilor în Java constă din trio-ul *`try-throw-catch`*
- Blocul `try` conține codul pentru algoritmul implementat
 - Acest cod spune ce se face atunci când totul merge bine
 - Se numește bloc `try` deoarece el "încearcă" să execute cazul în care totul merge așa cum a fost planificat
 - De asemenea acest bloc poate conține cod care aruncă o excepție dacă se întâmplă ceva neobișnuit

```
try {  
    CodCarePoateAruncaOExcepție  
}
```



Tratarea excepțiilor.

Mecanismul `try-throw-catch`

- Aruncarea explicită a unei excepții

```
throw new NumeleClaseiExcepție(PosibilArgumente);
```

- La aruncarea unei excepții, execuția blocului `try` în care a fost aruncată excepția se oprește
 - Normal, controlul este transferat unei alte porțiuni de cod, blocul `catch` (blocul de interceptare)
- Valoarea aruncată este argumentul operatorului `throw`; ea este întotdeauna un obiect aparținând unei clase excepție
 - Execuția unei instrucțiuni `throw` se numește *aruncare a unei excepții*



Tratarea excepțiilor.

Mecanismul `try-throw-catch`

- O instrucțiune `throw` seamănă cu un apel de metodă
`throw new NumeClasaExceptie(UnString) ;`
 - În exemplul de mai sus, obiectul de clasa `NumeClasaExceptie` este creat folosind ca argument un șir de caractere
 - Acest obiect, care este argument pentru operatorul `throw`, este obiectul excepție aruncat
- În loc să apeleze o metodă, instrucțiunea `throw` apelează un bloc `catch`



Tratarea excepțiilor.

Mecanismul `try-throw-catch`

- La aruncarea unei excepții se începe executarea blocului **catch**
 - Blocul **catch** are *un parametru*
 - Obiectul excepție aruncat este transmis ca parametru al blocului **catch**
- Un bloc **catch** este o porțiune de cod separată care se execută atunci când un program întâlnește și execută o instrucțiune **throw** în blocul **try** precedent
 - Execuția blocului **catch** se numește *interceptarea/"prinderea" excepției, sau tratarea excepției*

```
catch (Exception e) {  
    CodDeTratareAExcepției  
}
```



Tratarea excepțiilor.

Mecanismul `try-throw-catch`

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

- Identificatorul `e` din blocul `catch` de deasupra se numește parametru al blocului `catch`
- Parametrul blocului `catch` îndeplinește două roluri:
 1. Specifică tipul de obiect excepție aruncat pe care blocul `catch` îl poate intercepta (d.e., mai sus este un obiect de clasă `Exception`)
 2. Oferă un nume (pentru obiectul care este interceptat) care să fie folosit în blocul `catch`
 - Observație: adesea se folosește identificatorul `e` prin convenție, dar se poate folosi orice identificator care nu este cuvânt cheie



Tratarea excepțiilor.

Mecanismul `try-throw-catch`

- La executarea unui bloc `try` se pot întâmpla două lucruri:
 1. Nu este aruncată nici o excepție în blocul `try`
 - Codul din blocul `try` este executat până la sfârșitul blocului
 - Blocul `catch` este sărit
 - Execuția continuă de la codul amplasat după blocul `catch`
 2. Este aruncată o excepție în blocul `try` și interceptată în blocul `catch`
 - Restul codului din blocul `try` este sărit
 - Controlul se transferă la un bloc `catch` următor (în cazurile simple)
 - Obiectul aruncat este transmis ca parametru al blocului `catch`
 - Se execută codul din blocul `catch`
 - Se execută codul care urmează după blocul `catch` respectiv (dacă există)



Blocuri `catch` multiple

- Un bloc `try` poate arunca potențial orice număr de valori excepție, iar acestea pot fi de tipuri diferite
 - În oricare execuție a unui bloc `try`, poate fi aruncată cel mult o excepție (de vreme ce instrucțiunea `throw` termină execuția blocului `try`)
 - La execuții diferite ale blocului `try` pot fi aruncate valori diferite
- Fiecare bloc `catch` poate intercepta valorile de tipul de clasă excepție, date în antetul blocului `catch`
- Se pot intercepta tipuri diferite de excepții punând mai multe blocuri `catch` după un bloc `try`
 - Se pot pune oricâte blocuri `catch`, dar în ordinea corectă



Capcană: Interceptați mai întâi cea mai specifică excepție

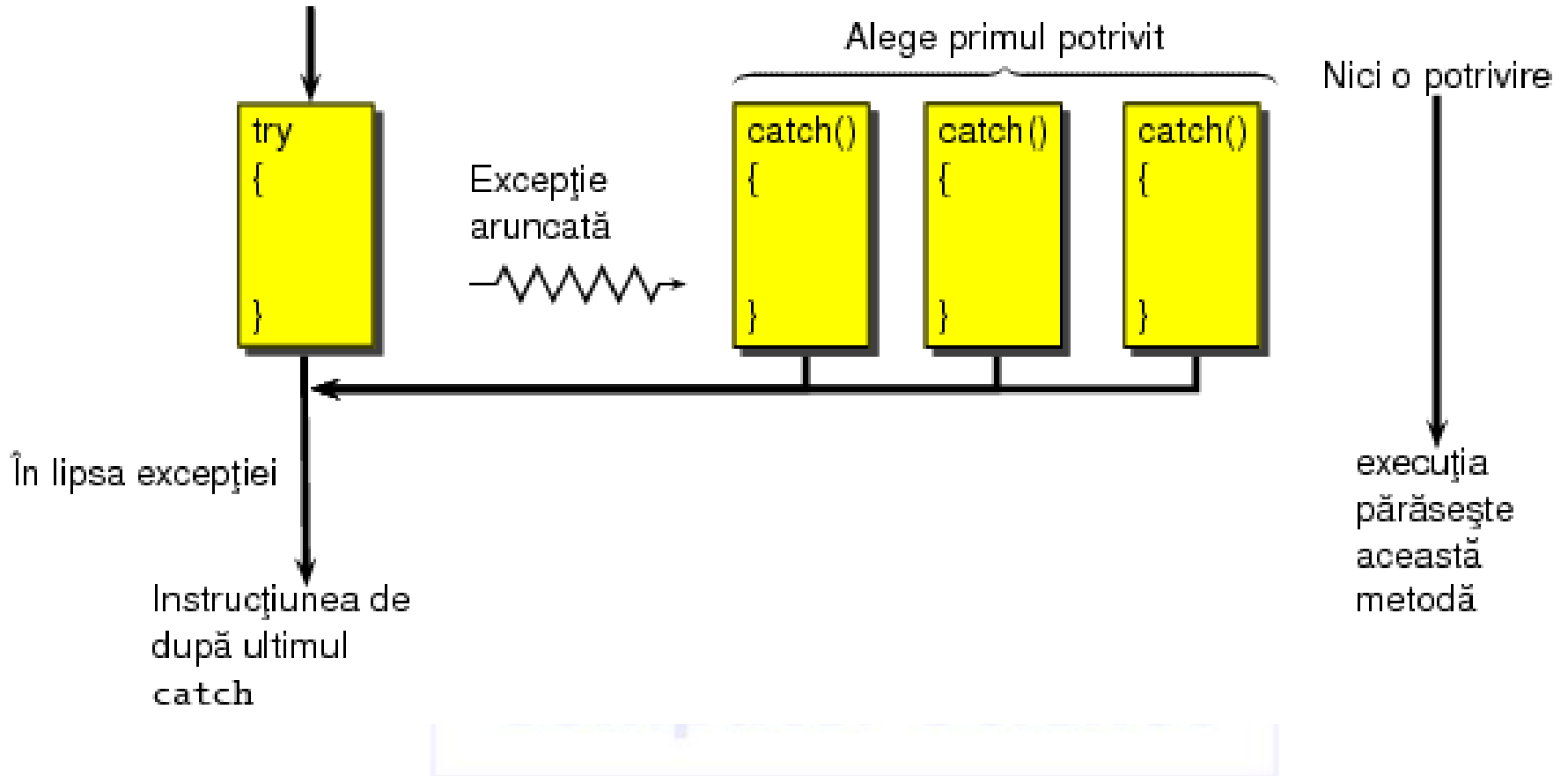
- La interceptarea de excepții multiple, ordinea blocurilor `catch` este importantă
 - **La aruncarea unei excepții într-un bloc `try`, blocurile `catch` sunt examinate în ordinea apariției**
 - **Este executat primul bloc care se potrivește cu tipul de excepție aruncat**

```
catch (Exception e)
{ . . . }
catch (NumberFormatException e)
{ . . . }
```
- Deoarece `NumberFormatException` este un tip de `Exception`, toate `NumberFormatExceptions` vor fi interceptate de către primul bloc `catch` înainte de a ajunge vreodată la cel de-al doilea
 - **Blocul `catch` pentru `NumberFormatException` nu va fi folosit!**
- Pentru ordine corectă, inversați cele două blocuri



Tratarea excepțiilor.

Mecanismul `try-throw-catch`





Exemplu cu două excepții

```
public class DublaGreseala {
    public static void main(String[] args) {
        int num = 5, denom = 0, result;
        int[] arr = {7, 21, 31};
        try
        {
            result = num / denom;
            result = arr[num];
        }
        catch (ArithmeticException ex) {
            System.out.println("Eroare aritmetica");
        }
        catch (IndexOutOfBoundsException ex) {
            System.out.println("Eroare de indice");
        }
    }
}
```

Observație:
Cea de a doua
excepție nu va fi
aruncată niciodată!



Tratarea excepțiilor.

Mecanismul `try-throw-catch`

- La aruncarea unei excepții de către o instrucțiune blocul `try{}`, blocurile `catch{}` sunt examinate unul câte unul începând cu primul
- Un singur bloc `catch{}` este ales
- Dacă nici un bloc `catch{}` nu se potrivește cu excepția, atunci nu este ales nici unul, iar execuția părăsește metoda respectivă (exact ca în lipsa blocului `catch{}`)
- Primul bloc `catch{}` care se potrivește cu tipul de excepție aruncată obține controlul
- Cele mai specifice tipuri de excepție trebuie să apară la început, urmate de tipurile mai generale de excepție
- Instrucțiunile din blocul `catch{}` ales sunt executate secvențial; după executarea ultimei instrucțiuni, controlul ajunge la prima instrucțiune care urmează după structura `try/catch`
- Controlul nu se întoarce în blocul `try`



Exemplu de intrare "prietenoașă"

```
import java.lang.* ;
import java.io.* ;

public class SquareUser
{
    public static void main ( String[] a )
        throws IOException
    {
        BufferedReader stdin =
            new BufferedReader (
                new InputStreamReader(System.in));
        String  inData = null;
        int      num = 0;
        boolean inputOK = false;
        while ( !inputOK )
        {
            System.out.print("Introduceti un
                               intreg:");

            inData = stdin.readLine();

            try
            {
                num = Integer.parseInt( inData );
                inputOK = true;
            }
            catch (NumberFormatException ex )
            {
                System.out.println("Ati introdus
                                     date invalide.");
                System.out.println("Va rog sa
                                     reincercati.\n");
            }
        }
    }
}

System.out.println("Patratul lui " +
                    inData + " este " + num*num);
} //end while
```



Clauza **finally**

- Excepția provoacă terminarea metodei curente
- Pericol: se poate sări peste o porțiune de cod esențială
- Exemplu

```
reader = new FileReader(filename);  
Scanner in = new Scanner(reader);  
readData(in);  
reader.close(); // s-ar putea sa nu ajunga aici niciodata
```

- Trebuie executat **reader.close()** chiar dacă apare o excepție
- Folosim clauza **finally** pentru codul care trebuie executat "indiferent de ce se întâmplă" (necondiționat)



Blocuri `catch` multiple și clauza `finally`

- Dacă există clauze `catch` asociate blocului `try`, atunci trebuie să punem clauza `finally` după toate clauzele `catch`

```
try {  
    // Bloc de cod cu puncte de iesire multiple  
}  
catch (OneException e) {  
    System.out.println(" Am interceptat OneException!");  
}  
catch (OtherException e) {  
    System.out.println(" Am interceptat OtherException!");  
}  
catch (AnotherException e) {  
    System.out.println(" Am interceptat AnotherException!");  
}  
finally {  
    // Bloc de cod executat intotdeauna la iesirea din bloc,  
    // indiferent de cum s-a iesit din "try"  
    System.out.println("Finally este executat intotdeauna");  
}
```



Clase excepție

- Există mai multe clase excepție pe lângă clasa **Exception**
 - Există mai multe clase excepție în bibliotecile standard Java
 - Pot fi definite noi clase excepție exact ca orice alte clase
- Toate clasele excepție predefinite au următoarele proprietăți
 - Posedă un constructor cu un singur argument de tipul **String**
 - Clasa are o metoda accesoare, **getMessage()**, care poate recupera șirul dat ca argument constructorului la crearea obiectului excepție
- Toate clasele excepție definite de programator ar trebui să aibă aceleași proprietăți



Clase excepție din pachetele standard

- Există numeroase clase excepție predefinite care sunt incluse în pachetele standard Java:
 - `IOException`
 - `NoSuchMethodException`
 - `FileNotFoundException`
- Multe clase excepție trebuie importate pentru a le putea utiliza:

```
import java.io.IOException;
```
- Clasa predefinită **Exception** este clasa rădăcină pentru toate excepțiile
 - Fiecare clasă excepție este descendentă din clasa **Exception**
 - Clasa **Exception** poate fi folosită: direct sau pentru a defini o clasă derivată
 - Clasa **Exception** se află în pachetul `java.lang` și nu trebuie clauză `import`



Folosirea metodei `getMessage`

```
. . . // codul metodei
try
{
. . .
throw new Exception(StringArgument);
. . .
}
catch(Exception e)
{
String message = e.getMessage();
System.out.println(message);
System.exit(0);
} . . .
```

- Fiecare excepție are o variabilă instanță de tipul **String** care conține un mesaj
 - Acest șir identifică de obicei motivul apariției excepției
- **StringArgument** este folosit ca valoare pentru variabila instanță de tip șir a excepției **e**
 - De aceea, apelul de metodă **e.getMessage()** returnează acest șir



Definirea claselor excepție

- Fiecare clasă excepție care urmează să fie definită trebuie să fie o clasă derivată dintr-o clasă excepție deja definită
 - Derivată din oricare clasă excepție definită în bibliotecile standard Java sau definită de către programator
- Constructorii sunt membrii cei mai importanți în definirea unei clase excepție
 - Constructorii trebuie să se comporte corespunzător în raport cu variabilele și metodele moștenite din clasa de bază
 - Adesea, nu există alți membri cu excepția celor moșteniți din clasa de bază
- Clasa care urmează nu efectuează decât aceste sarcini fundamentale



O clasă excepție definită de către programator

```
public class DivisionByZeroException extends Exception
{
    public DivisionByZeroException()
    {
        super("Division by zero.");
    }
    public DivisionByZeroException(String message)
    {
        super(message);
    }
}
```

/* Se poate face mai mult într-un constructor de excepție, dar aceasta este o formă uzuală */

/* super invocă constructorul clasei de bază Exception */



Caracteristicile obiectului Exception

- Cele mai importante două lucruri referitoare la un obiect excepție sunt tipul său (adică, clasa excepție) și mesajul pe care îl poartă
 - Mesajul este transmis împreună cu obiectul excepție ca variabilă instanță
 - Acest mesaj poate fi recuperat cu metoda accesoare `getMessage`, astfel că blocul `catch` poate folosi mesajul

OF CLUJ-NAPOCA
Computer Science



Indicații pentru clasele excepție definite de programator

- Clasele excepție pot fi definite de către programator, dar fiecare asemenea clasă trebuie să fie derivată dintr-o clasă excepție existentă deja
- Clasa **Exception** poate fi folosită pe post de clasă de bază, cu excepția cazului în care o altă clasă excepție este mai potrivită
- Trebuie definiți cel puțin doi constructori, iar uneori mai mulți
- Excepția trebuie să țină seama că metoda **getMessage ()** este moștenită



Să păstreze getMessage

- Pentru toate clasele excepție predefinite, **getMessage** returnează șirul de caractere transmis ca argument constructorului său
 - Sau să returneze un șir implicit dacă nu s-a transmis nici un argument constructorului
- Acest comportament trebuie păstrat în toate clasele excepție definite de către programator
 - Trebuie inclus un *constructor* care are *un parametru șir* de caractere și al cărui corp începe cu un apel la **super**
 - Apelul la **super** trebuie să folosească parametrul ca argument al său
 - Trebuie inclus și un *constructor fără argumente* al cărui corp începe cu un apel la **super**
 - Acest apel la **super** trebuie să folosească *șirul implicit* ca argument



Aruncarea unei excepții într-o metodă

- Uneori are sens să se arunce o excepție într-o metodă fără a o intercepta în metoda respectivă
 - Unele programe care folosesc o anumită metodă ar trebui să se termine pur și simplu la aruncarea unei excepții, iar altele nu
 - În astfel de cazuri, programul care folosește invocarea metodei ar trebui să o includă într-un bloc **try** și să intercepteze excepția într-un bloc **catch** care urmează
- În acest caz, metoda în sine nu va include blocuri **try** și **catch**
 - Totuși, trebuie să conțină o *clauză throws*



Declararea excepțiilor în clauza `throws`

- Dacă o metodă poate arunca o excepție, dar nu o interceptează, atunci ea trebuie să furnizeze un avertisment
 - Acest avertisment se numește *clauză `throws`*
 - Procesul de includere a unei clase excepție într-o clauză `throws` se numește *declararea excepției*
`throws OExcepție //clauza throws`
 - Următorul cod declară că invocarea lui `oMetoda` poate cauza aruncarea lui `OExcepție`
`public void oMetoda() throws OExcepție`
- `main` este o metodă care poate avea și ea specificarea unei excepții:

```
public static void main(String[] args) throws Exception
```



Declararea excepțiilor în clauza `throws`

- Dacă o metodă poate arunca mai mult de un fel de excepție, atunci tipurile se separă prin virgule
`public void oMetoda() throws OExcepție, AltaExcepție`
- Dacă o metodă aruncă o excepție și nu o interceptează, atunci apelul metodei se termină imediat

OF CLUJ-NAPOCA
Computer Science



Regula "prinde sau declară"

- Cele mai obișnuite excepții care ar putea fi aruncate într-o metodă trebuie tratate în unul dintre următoarele două moduri:
 1. Codul care poate arunca o excepție este pus într-un bloc **try**, iar excepția care poate apărea este interceptată într-un bloc **catch** din aceeași metodă
 2. Excepția posibilă poate fi declarată la începutul definiției metodei punând numele clasei excepție într-o clauză **throws**

Computer Science



Regula "prinde sau declară"

- Prima dintre tehnici tratează o excepție într-un bloc **catch**
- Cea de a doua tehnică este o modalitate de a deplasa răspunderea pentru tratarea excepției la metoda care a invocat-o pe cea care a aruncat excepția
- Metoda apelantă trebuie să trateze excepția, cu excepția cazului în care folosește aceeași tehnică de "pasare"
- Într-un sfârșit, fiecare excepție ar trebui interceptată de un bloc **catch** din vreo metodă care nu numai declară într-o clauză **throws** ci și interceptează clasa de excepție respectivă



Regula "prinde sau declară"

- În oricare metodă, ambele tehnici pot fi amestecate
 - Unele excepții pot fi interceptate, iar altele declarate în clauza `throws`
- Cu toate acestea, tehnicile menționate trebuie folosite consistent pentru o excepție dată
 - Dacă o excepție nu este declarată, atunci ea trebuie tratată în metodă
 - Dacă este declarată excepția, atunci responsabilitatea pentru tratarea ei este pasată unei alte metode care o apelează
 - Observați că dacă definiția unei metode include invocarea unei a doua metode, iar cea de a doua poate arunca o excepție și nu o interceptează, atunci prima metodă trebuie să o declare sau să o intercepteze



Excepții verificate și neverificate

- Excepțiile care sunt supuse regulii "prinde sau declară" sunt numite **excepții verificate**
 - Compilatorul verifică pentru a vedea dacă excepțiile sunt luate în considerare fie într-un bloc `catch`, fie într-o clauză `throws`
 - Clasele `Throwable`, `Exception`, precum și toți descendenții clasei `Exception` (excepție `RuntimeException`) constituie excepții verificate
- **Toate celelalte excepții sunt neverificate**
 - Clasele `Error` și `RuntimeException` și toate clasele care descind din ele constituie excepții *neverificate*
 - Aceste clase *nu sunt* supuse regulii "prinde sau declară"

Computer Science

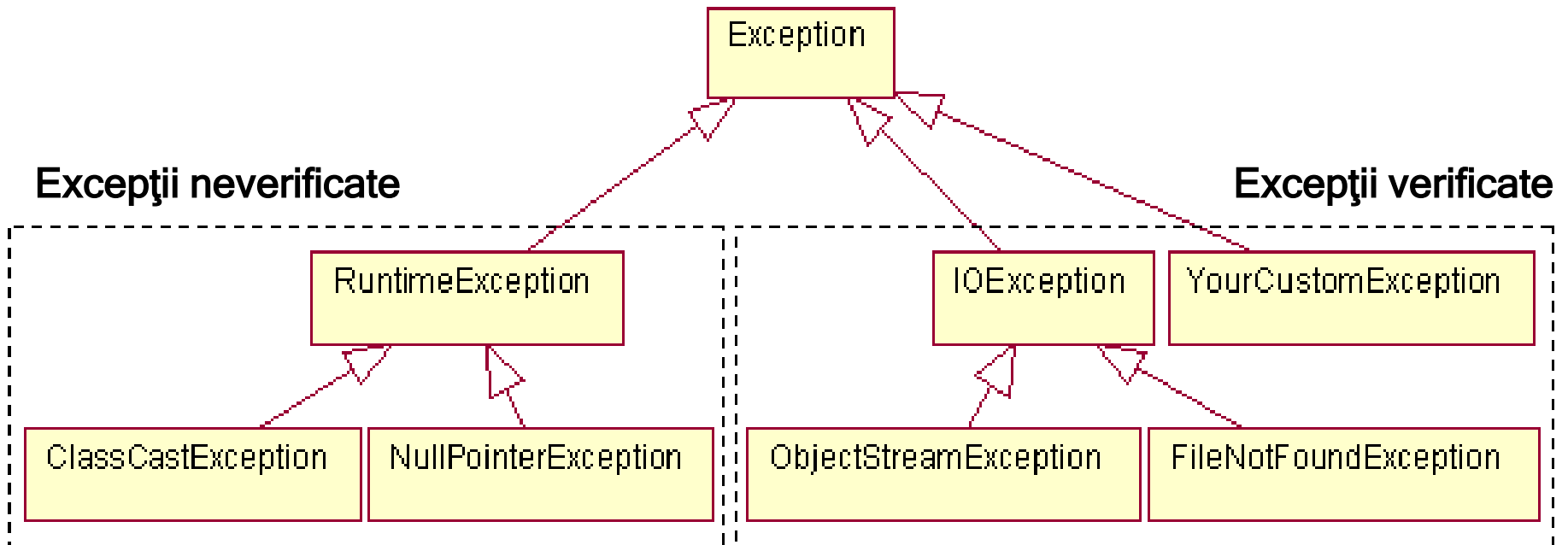


Excepții de la regula "prinde sau declară"

- Excepțiile *verificate* trebuie să respecte regula "prinde sau declară"
 - Programele în care pot fi aruncate aceste excepții nu se vor compila până când excepțiile respective nu sunt tratate corespunzător
- Excepțiile *neverificate* nu sunt supuse regulii "prinde sau declară"
 - Programele în care apar astfel de excepții trebuie pur și simplu corectate întrucât au erori de alt fel (dacă compilatorul semnalează erori)



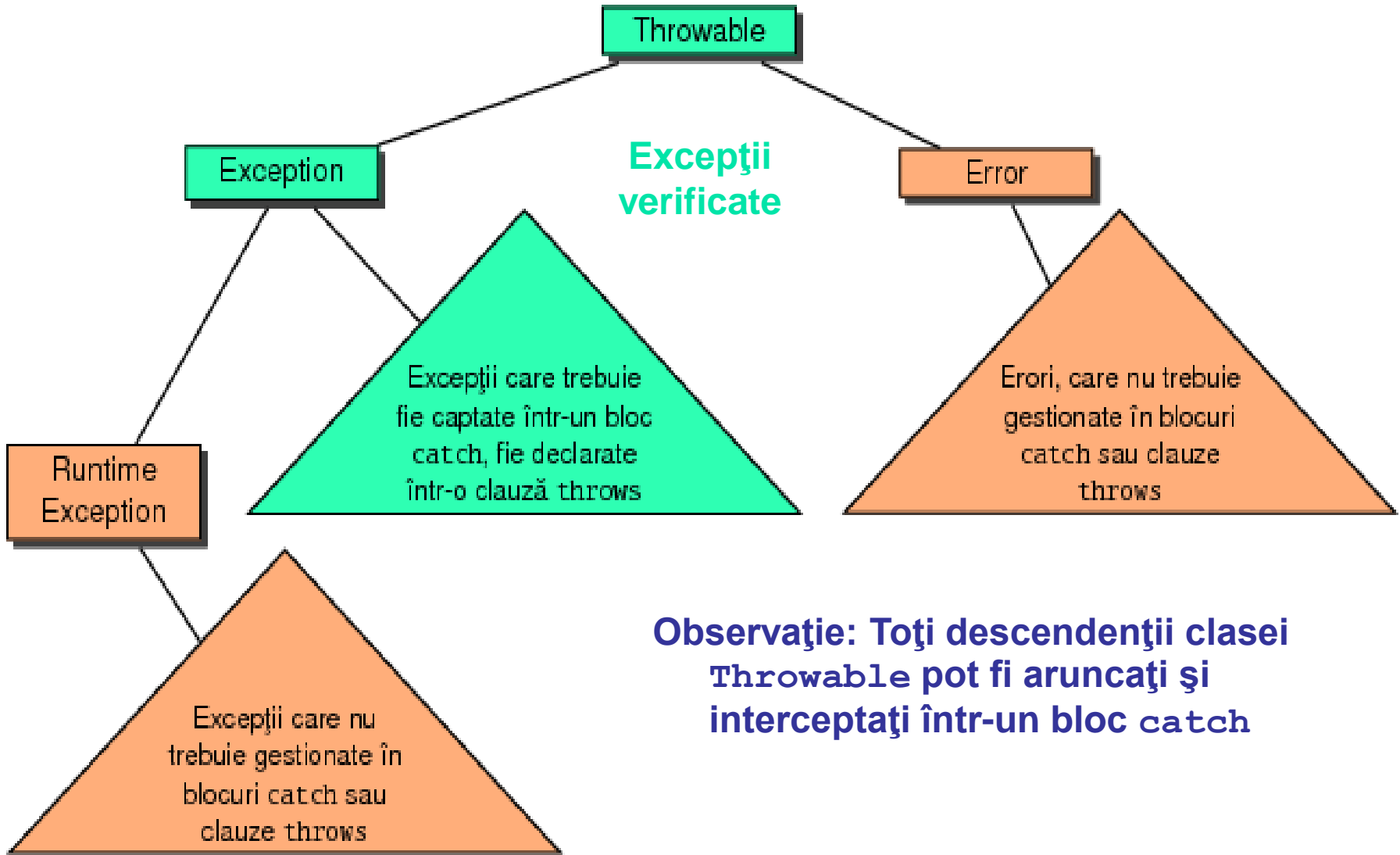
Excepții verificate și neverificate



Notă. Aici este o mică parte a ierarhiei!



Ierarhia obiectelor Throwable (aruncabile)



**Observație: Toți descendenții clasei
Throwable pot fi aruncați și
interceptați într-un bloc catch**



Clauza **throws** în clase derivate

- La suprascrierea unei metode într-o clasă derivată, aceasta trebuie să aibă aceleași clase excepție precum cele listate în clauza **throws** din clasa de bază
 - sau un subset al acestora
- O clasă derivată *nu poate adăuga* excepții la clauza **throws**
 - dar poate șterge câteva

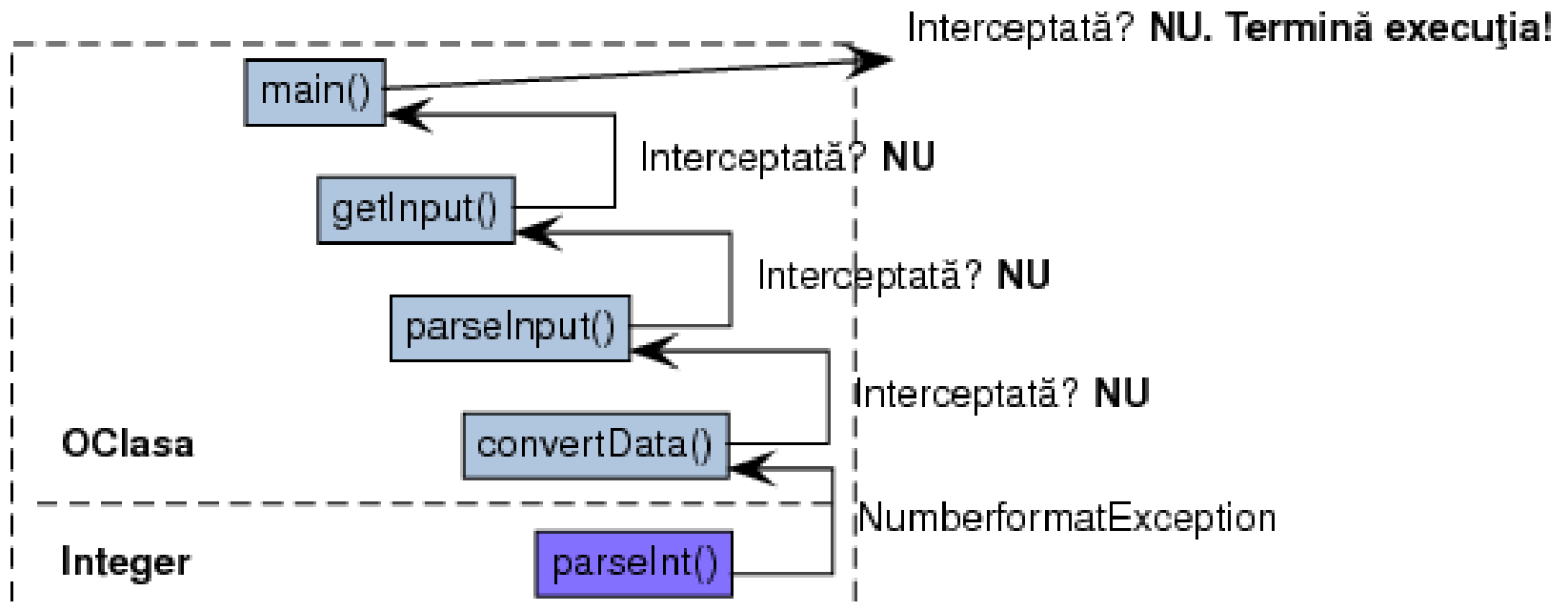


Ce se întâmplă dacă o excepție nu este interceptată?

- Dacă fiecare metodă până la, și inclusiv, metoda **main** include o clauză **throws**, excepția respectivă poate fi aruncată, dar poate să nu fie interceptată niciodată
 - Într-un program GUI (adică un program cu o interfață cu ferestre, grafică) nu se întâmplă nimic – atâta doar că utilizatorul poate fi lăsat într-o situație ne-explicată, iar programul poate să nu mai fie sigur
 - În programe non-GUI, aceasta face ca programul să se termine cu un mesaj de eroare care dă numele clasei excepție
- Fiecare program bine scris trebuie în cele din urmă să intercepteze fiecare excepție printr-un bloc **catch** în una dintre metodele sale



Propagarea excepției



Computer Science



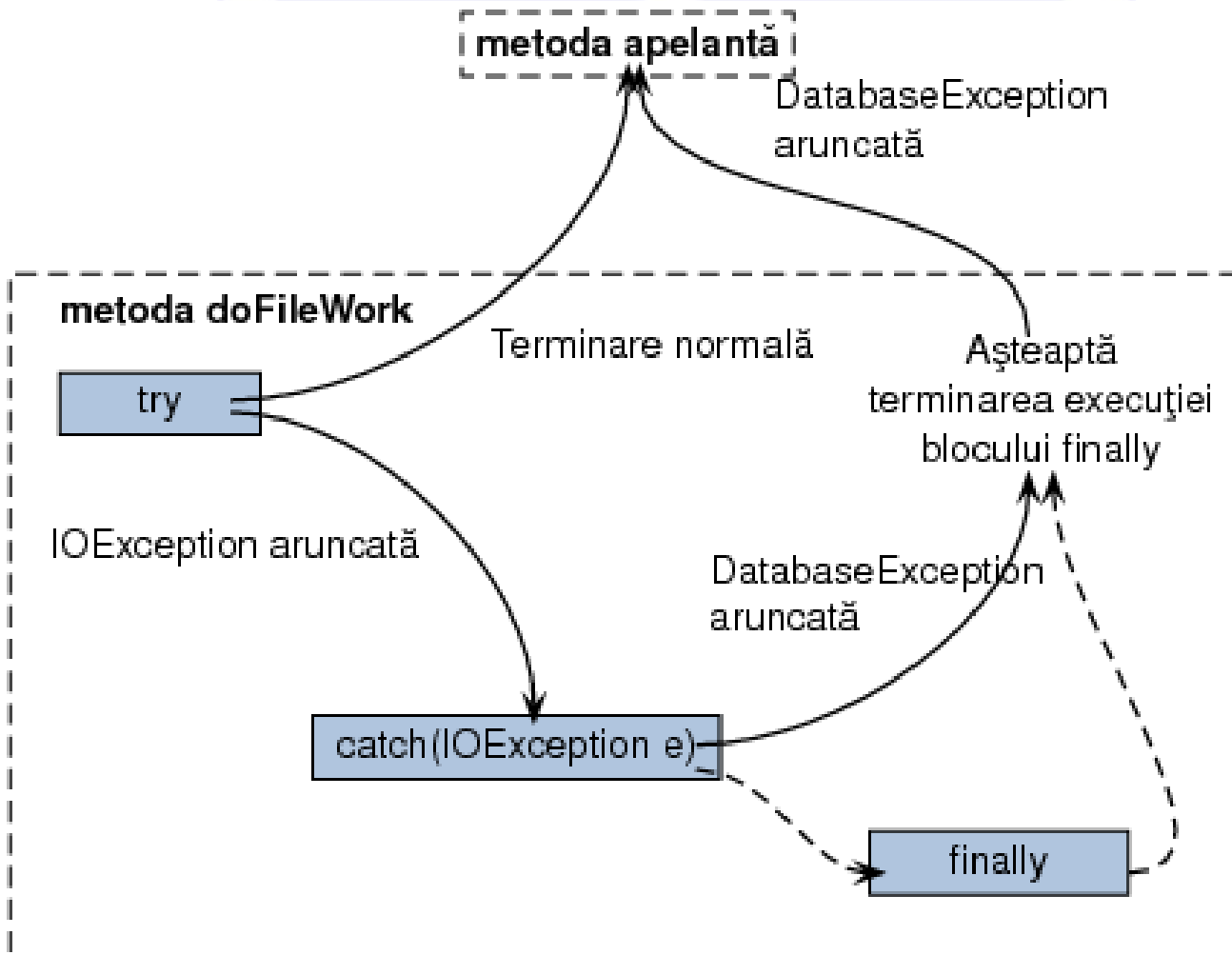
Un alt exemplu

```
public void doFileWork(String filename)
    throws DatabaseException
{
    FileOutputStream fos = null;
    ObjectOutputStream oos = null;
    try{
        fos = new FileOutputStream(filename);
        oos = new ObjectOutputStream(fos);
        oos.writeObject(obj);
    }
    catch(IOException e){
        throw new DatabaseException(
            "Problem while working with "
            +filename+": "+ e.getMessage());
    }
}

finally{
    try{
        if(oos!=null){
            oos.close();
        }
        if(fos!=null){
            fos.close();
        }
    }
    catch(IOException e){
        throw new DatabaseException(
            "Problem while working with "
            +filename+": "+e.getMessage());
    }
}
```



Explicații pentru exemplu





Când să folosim excepțiile

- Excepțiile trebuie rezervate pentru situațiile în care o metodă întâlnește *un caz neobișnuit sau neașteptat, care nu poate fi tratat cu ușurință în vreun alt mod*
- Atunci când trebuie folosită tratarea excepțiilor, folosiți aceste recomandări:
 - Includeți instrucțiuni **throw** și precizați clasele excepție într-o clauză **throws** din definiția metodei respective
 - Plasați blocurile **try** și **catch** într-o metodă diferită

Computer Science



Când să folosim excepțiile

- Iată un exemplu de metodă din care este aruncată o excepție:

```
public void oMetoda() throws OExcepție
{
    . . .
    throw new OExcepție(UnArgument);
    . . .
}
```

- Atunci când `oMetoda` este folosită de `altaMetoda`, `altaMetoda` trebuie să trateze excepția:

```
public void altaMetoda()
{
    try {
        oMetoda();
        . . .
    }
    catch (OExcepție e) {
        CodPentruTratareaExcepției
    }
    . . .
}
```




Ghid pentru excepții

- Dacă metoda întâlnește o condiție anormală pe care nu o poate trata, atunci trebuie să arunce o excepție
- Evitați folosirea excepțiilor pentru a indica situații care pot fi așteptate ca parte a funcționării normale a metodei
- Dacă metoda descoperă că clientul și-a încălcat obligațiile contractuale (spre exemplu, prin transmiterea de date de intrare neconforme specificației), atunci aruncați o excepție neverificată



Ghid pentru excepții

- Dacă metoda nu-și poate îndeplini contractul, atunci aruncați fie o excepție verificată, fie una neverificată
- Dacă aruncați o excepție pentru o situație anormală despre care considerați că programatorii trebuie să decidă în mod conștient cum să o trateze, atunci aruncați o excepție verificată
- Definiți sau alegeți o clasă excepție care există deja pentru fiecare fel de condiție anormală care poate face ca metoda dvs. să arunce o excepție



Re-aruncarea excepțiilor

- După interceptarea unei excepții, ea poate fi re-aruncată dacă e cazul
- La re-aruncarea unei excepții putem alege locația din care se va vedea ca aruncat obiectul în trasarea stivei de execuție
 - Putem face ca excepția re-aruncată să pară a fi aruncată din locul excepției originale sau din locul re-aruncării
- Pentru a re-arunca o excepție cu indicarea locației originale, pur și simplu o aruncăm din nou:

```
try {  
    cap(0);  
}  
catch(ArithmeticException e) {  
    throw e;  
}
```



Re-aruncarea excepțiilor

- Pentru a avea locația reală din care a fost re-aruncată apelăm metoda `fillInStackTrace()` a excepției
 - Metoda setează informația din trasarea stivei pe baza contextului de execuție curent. Exemplu:

```
try {  
    cap(0);  
}  
catch(ArithmeticException e) {  
    throw (ArithmeticException)e.fillInStackTrace();  
}
```

- Apelăm `fillInStackTrace()` pe linia cu instrucțiunea `throw` – astfel numărul de linie din trasare este la fel cu cel unde apare instrucțiunea `throw`
 - Metoda `fillInStackTrace()` returnează o referință la clasa `Throwable`, așa că e nevoie de o conversie de tip la tipul real de excepție



Exemplu de apel al metodei `fillInStackTrace()`

```
import java.lang.*;

public class ThrowableDemo {

    public static void function1() throws Exception {
        throw new Exception("this is thrown from
                               function1()");
    }

    public static void function2() throws Throwable{
        try {
            function1();
        }
        catch(Exception e) {
            System.err.println("Inside function2():");
            e.printStackTrace();
            throw e.fillInStackTrace();
        }
    }
}
```

```
public static void main(String[]
    args) throws Throwable {
    try {
        function2();
    }
    catch (Exception e) {
        System.err.println("Caught
                               Inside Main:");
        e.printStackTrace();
    }
}
```



Exemplu de apel al metodei `fillInStackTrace()`

- Rezultatul execuției acestui exemplu este:

Inside function2():

```
java.lang.Exception: this is thrown from function1()  
at ThrowableDemo.function1(ThrowableDemo.java:4)  
at ThrowableDemo.function2(ThrowableDemo.java:9)  
at ThrowableDemo.main(ThrowableDemo.java:19)
```

Caught Inside Main:

```
java.lang.Exception: this is thrown from function1()  
at ThrowableDemo.function2(ThrowableDemo.java:13)  
at ThrowableDemo.main(ThrowableDemo.java:19)
```