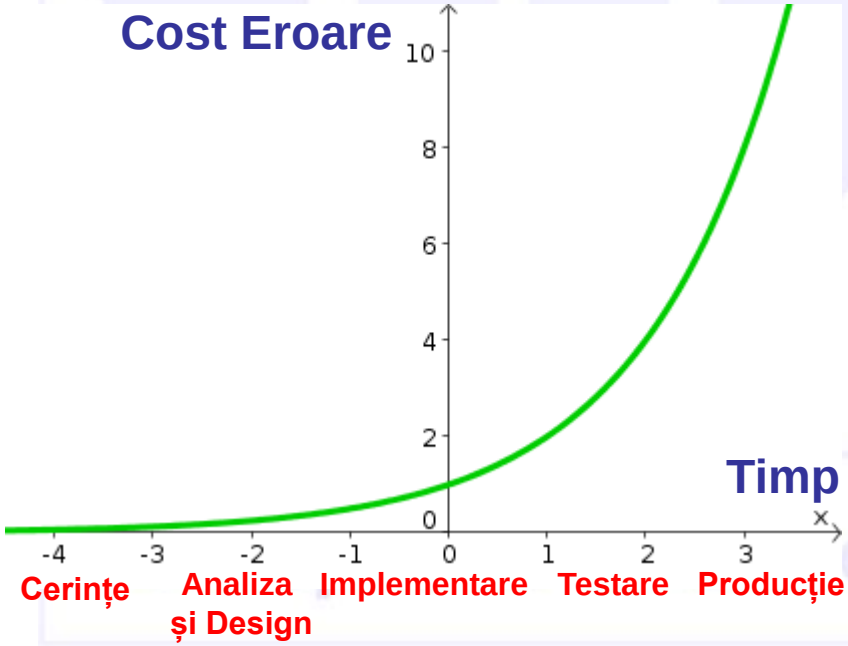
Programare orientată pe obiecte

1. Erori şi excepţii în Java

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

Aserţiuni

- 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**

Aserţiuni

- Sintaxa:
  - assert expr1;

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

    sau
  - assert expr1 : expr2

    expr1 – expresie booleană
    expr2 – valoare, mesajul erorii

Activarea/dezactivarea aserţiunilor

- Activare:

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

- Dezactivare:

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

- Observaţie: în alte medii de dezvoltare, opţiunea de utilizare a aserţiunilor trebuie activată/dezactivată explicit
  - În Eclipse opţiunile de mai sus sunt introduse prin accesarea Run Configurations -> tab-ul Arguments -> căsuţa “VM Arguments”, unde se inserează -ea sau -da

Aserţiuni: 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");

}
```

Aserţiuni: 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

Unde se pun aserţiunile

- Locuri posibile
  - **Precondiţia** metodei – se verifică ce trebuie să fie adevărat atunci când se execută o metodă
  - **Postcondiţia** metodei – ce trebuie să fie adevărat după ce s-a executat metoda
  - **Invarianți 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

Asertiuni: 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

- Invarianți interni

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

- Invarianți 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 ...;
```

Excepţii. Probleme în cursul execuţiei programelor

- Un program întâlneşte adesea probleme (**excepții**) în cursul execuţiei sale:
  - poate avea probleme la citirea datelor,
  - pot exista caractere nepermise în date sau
  - indexul unui tablou poate depăşi limitele acestuia
- Excepţiile Java permit programatorului să trateze astfel de probleme
  - Putem scrie programe care îşi revin la întâlnirea excepțiilor şi îşi continuă execuţia
  - **Programele nu trebuie să eşueze atunci când utilizatorul face o greşeală!**
- În special intrarea şi ieşirea sunt susceptibile la excepții
- Tratarea excepţiilor este esenţială pentru programarea I/E

Exemplu de apariţie a unei excepţii

- 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)
```

Discuţie asupra exemplului

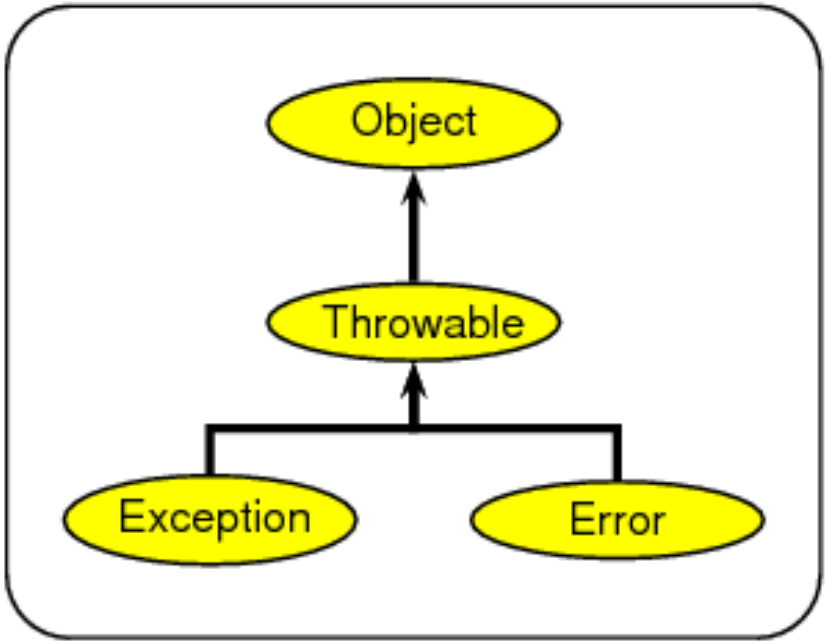
- Programul nu este greşit
  - Problema este că **`nextInt`** nu poate converti şirul de caractere "**`h1`**" la un **`int`**
  - În momentul în care **`nextInt`** a întâlnit problema, metoda a **aruncat** o excepţie de tipul **`InputMismatchException`**
  - Sistemul de execuţie Java a interceptat (a "prins") excepţia, a oprit programul şi a tipărit mesajele de eroare

Excepţii şi erori

- O **excepţie:** o problemă care apare în cursul execuţiei unui program
  - La apariţia unei excepţii, JVM creează un obiect de clasa `Exception` care conţine informaţii despre problema apărută
  - Însuşi programul Java poate **intercepta** (**catch**) o excepţie. Apoi poate folosi obiectul de tipul excepţie pentru a-şi reveni după problemă
- Şi 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-şi ***înceteze execuţia***

Ierarhia Throwable ("aruncabil")

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



Tratarea excepțiilor. Mecanismul `try-throw-catch`

- Calea fundamentală pentru tratarea excepţiilor în Java constă din trio-ul ***`try-throw-catch`***

- Blocul `try` conţine codul pentru algoritmul implementat
  - Acest cod spune ce se face atunci când totul merge bine
  - Se numeşte bloc `try` deoarece el "încearcă" să execute cazul în care totul merge aşa cum a fost planificat
  - De asemenea acest bloc poate conţine cod care aruncă o excepţie dacă se întâmplă ceva neobişnuit

```
try {
   CodCarePoateAruncaOExceptie
}
```

Tratarea excepțiilor. Mecanismul `try-throw-catch`

- Aruncarea explicită a unei excepții

```
throw new NumeleClaseiExceptie(PosibilArgumente);
```

- La aruncarea unei excepţii, execuţia blocului **`try`** în care a fost aruncată excepţia se opreşte
  - Normal, controlul este transferat unei alte porţiuni de cod, blocul **`catch`** (blocul de interceptare)
- Valoarea aruncată este argumentul operatorului **`throw`**; ea este întotdeauna un obiect aparţinând unei clase excepţie
  - Execuţia unei instrucţiuni **`throw`** se numeşte *aruncare a unei excepţii*

Tratarea excepțiilor. Mecanismul `try-throw-catch`

- O instrucţiune **`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 şir de caractere
  - Acest obiect, care este argument pentru operatorul **`throw`**, este obiectul excepţie aruncat
- În loc să apeleze o metodă, instrucţiunea **`throw`** apelează un bloc **`catch`**

Tratarea excepțiilor. Mecanismul `try-throw-catch`

- La aruncarea unei excepţii se începe executarea blocului **`catch`**
  - Blocul **`catch`** are *un parametru*
  - Obiectul excepţie aruncat este transmis ca parametru al blocului **`catch`**
- Un bloc **`catch`** este o porţiune de cod separată care se execută atunci când un program întâlneşte şi execută o instrucţiune **`throw`** în blocul **`try`** precedent
  - Execuţia blocului **`catch`** se numeşte *interceptarea/"prinderea" excepţiei*, sau *tratarea excepţiei*

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

Tratarea excepțiilor. Mecanismul `try-throw-catch`

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

- Identificatorul `e` din blocul `catch` de deasupra se numeşte parametru al blocului `catch`

- Parametrul blocului `catch` îndeplineşte două roluri:
  1. Specifică tipul de obiect excepţie aruncat pe care blocul `catch` îl poate intercepta (d.e., mai sus este un obiect de clasa `Exception`)
  2. Oferă un nume (pentru obiectul care este interceptat) care să fie folosit în blocul `catch`
     - Observaţie: adesea se foloseşte identificatorul `e` prin convenţie, 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 excepţie în blocul **`try`**
     - Codul din blocul **`try`** este executat până la sfârşitul blocului
     - Blocul **`catch`** este sărit
     - Execuţia continuă de la codul amplasat după blocul **`catch`**
  2. Este aruncată o excepţie în blocul **`try`** şi 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 potenţial orice număr de valori excepţie, iar acestea pot fi de tipuri diferite
  - În oricare execuţie a unui bloc `try`, poate fi aruncată cel mult o excepţie (de vreme ce instrucţiunea `throw` termină execuţia blocului `try`)
  - La execuţii diferite ale blocului `try` pot fi aruncate valori diferite
- Fiecare bloc `catch` poate intercepta valorile de tipul de clasă excepţie, date în antetul blocului `catch`
- Se pot intercepta tipuri diferite de excepţii punând mai multe blocuri `catch` după un bloc `try`
  - Se pot pune oricâte blocuri `catch`, dar în ordinea corectă

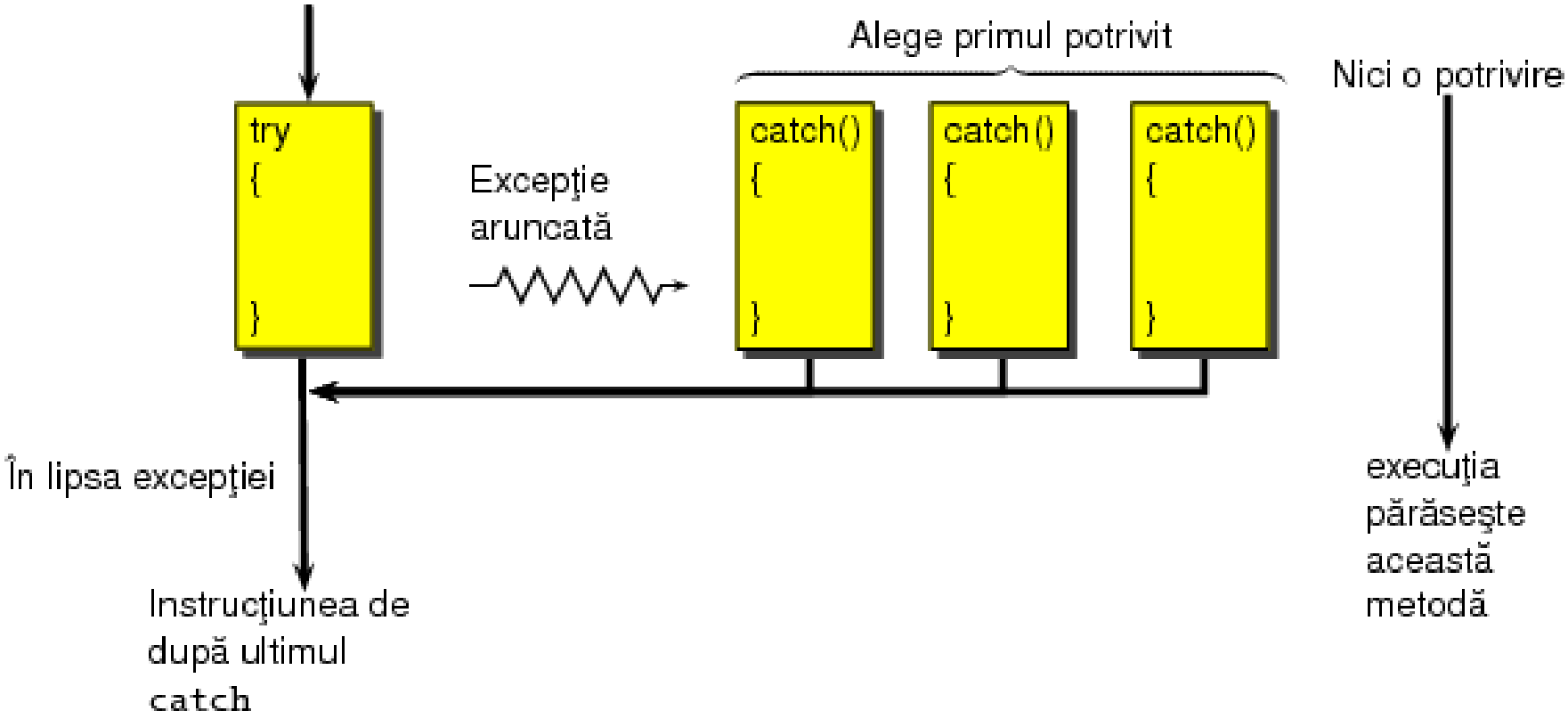
Capcană: Interceptaţi mai întâi cea mai specifică excepţie

- La interceptarea de excepţii multiple, ordinea blocurilor `catch` este importantă
  - **La aruncarea unei excepţii într-un bloc `try`, blocurile `catch` sunt examinate în ordinea apariţiei**
  - **Este executat primul bloc care se potriveşte cu tipul de excepţie aruncat**

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

- Deoarece `NegativeNumberException` este un tip de `Exception`, toate `NegativeNumberExceptions` vor fi interceptate de către primul bloc `catch` înainte de a ajunge vreodată la cel de-al doilea
  - **Blocul catch pentru `NegativeNumberException` nu va fi folosit!**
- Pentru ordine corectă, inversaţi cele două blocuri

Tratarea excepțiilor. Mecanismul `try-throw-catch`



Exemplu cu două excepţii

```
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");
    }
  }
}
```

Observaţie:
Cea de a doua excepţie nu va fi aruncată niciodată!

Tratarea excepțiilor. Mecanismul `try-throw-catch`

- La aruncarea unei excepţii de către o instrucţiune 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 potriveşte cu excepţia, atunci nu este ales nici unul, iar execuţia părăseşte metoda respectivă (exact ca în lipsa blocului **catch{}**)
- Primul bloc **catch{}** care se potriveşte cu tipul de excepţie aruncată obţine controlul
- Cele mai specifice tipuri de excepţie trebuie să apară la început, urmate de tipurile mai generale de excepţie
- Instrucţiunile din blocul **catch{}** ales sunt executate secvenţial; după executarea ultimei instrucţiuni, controlul ajunge la prima instrucţiune care urmează după structura **try/catch**
- Controlul *nu se întoarce* în blocul **try**

Exemplu de intrare "prietenoasă"

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

Clauza `finally`

- Excepţia provoacă terminarea metodei curente
- Pericol: se poate sări peste o porţiune de cod esenţială
- 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 excepţie
- Folosim clauza `finally` pentru codul care trebuie executat "indiferent de ce se întâmplă" (necondiţionat)

Blocuri `catch` multiple şi 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 excepţie

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

Clase excepţie din pachetele standard

- Există numeroase clase excepţie predefinite care sunt incluse în pachetele standard Java:

```
IOException
NoSuchMethodException
FileNotFoundException
```

  - Multe clase excepţie trebuie importate pentru a le putea utiliza:

```
import java.io.IOException;
```

- Clasa predefinită **`Exception`** este clasa rădăcină pentru toate excepţiile
  - Fiecare clasă excepţie 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`** şi 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 excepţie are o variabilă instanţă de tipul `String` care conţine un mesaj
  - Acest şir identifică de obicei motivul apariţiei excepţiei
- `StringArgument` este folosit ca valoare pentru variabila instanţă de tip şir a excepţiei `e`
  - De aceea, apelul de metodă `e.getMessage()` returnează acest şir

Definirea claselor excepţie

- Fiecare clasă excepţie care urmează să fie definită trebuie să fie o clasă derivată dintr-o clasă excepţie deja definită
  - Derivată din oricare clasă excepţie definită în bibliotecile standard Java sau definită de către programator
- Constructorii sunt membrii cei mai importanţi în definirea unei clase excepţie
  - Constructorii trebuie să se comporte corespunzător în raport cu variabilele şi metodele moştenite din clasa de bază
  - Adesea, nu există alţi membri cu excepţia celor moşteniţi din clasa de bază
- Clasa care urmează nu efectuează decât aceste sarcini fundamentale

O clasă excepţie 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 excepţie, 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 excepţie sunt tipul său (adică, clasa excepţie) şi mesajul pe care îl poartă
  - Mesajul este transmis împreună cu obiectul excepţie ca variabilă instanţă
  - Acest mesaj poate fi recuperat cu metoda accesoare `getMessage`, astfel că blocul `catch` poate folosi mesajul

Indicaţii pentru clasele excepţie definite de programator

- Clasele excepţie pot fi definite de către programator, dar fiecare asemenea clasă trebuie să fie derivată dintr-o clasă excepţie existentă deja
- Clasa `Exception` poate fi folosită pe post de clasă de bază, cu excepţia cazului în care o altă clasă excepţie este mai potrivită
- Trebuie definiţi cel puţin doi constructori, iar uneori mai mulţi
- Excepţia trebuie să ţină seama că metoda `getMessage()` este moştenită

Să păstreze `getMessage`

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

Aruncarea unei excepţii într-o metodă

- Uneori are sens să se arunce o excepţie într-o metodă fără a o intercepta în metoda respectivă
  - Unele programe care folosesc o anume metodă ar trebui să se termine pur şi simplu la aruncarea unei excepţii, iar altele nu
  - În astfel de cazuri, programul care foloseşte invocarea metodei ar trebui să o includă într-un bloc **`try`** şi să intercepteze excepţia într-un bloc **`catch`** care urmează
- În acest caz, metoda în sine nu va include blocuri **`try`** şi **`catch`**
  - Totuşi, trebuie să conţină o *clauză* **`throws`**

Declararea excepţiilor în clauza `throws`

- Dacă o metodă poate arunca o excepţie, dar nu o interceptează, atunci ea trebuie să furnizeze un avertisment
  - Acest avertisment se numeşte *clauză* ***`throws`***
  - Procesul de includere a unei clase excepţie într-o clauză throws se numeşte *declararea excepţiei*

```
throws OExceptie  //clauza throws
```

  - Următorul cod declară că invocarea lui `oMetoda` poate cauza aruncarea lui `OExceptie`

```
public void oMetoda() throws OExceptie
```

- `main` este o metodă care poate avea şi ea specificarea unei excepţii:

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

Declararea excepţiilor în clauza `throws`

- Dacă o metodă poate arunca mai mult de un fel de excepţie, atunci tipurile se separă prin virgule

```
public void oMetoda() throws OExceptie, AltaExceptie
```

- Dacă o metodă aruncă o excepţie şi nu o interceptează, atunci apelul metodei se termină imediat

Regula "prinde sau declară"

- Cele mai obişnuite excepţii care ar putea fi aruncate într-o metodă trebuie tratate în unul dintre următoarele două moduri:

1. Codul care poate arunca o excepţie este pus într-un bloc **`try`**, iar excepţia care poate apărea este interceptată într-un bloc **`catch`** din aceeaşi metodă

2. Excepţia posibilă poate fi declarată la începutul definiţiei metodei punând numele clasei excepţie într-o clauză **`throws`**

Regula "prinde sau declară"

- Prima dintre tehnici tratează o excepţie într-un bloc **`catch`**
- Cea de a doua tehnică este o modalitate de a deplasa răspunderea pentru tratarea excepţiei la metoda care a invocat-o pe cea care a aruncat excepţia
- Metoda apelantă trebuie să trateze excepţia, cu excepţia cazului în care foloseşte aceeaşi tehnică de "pasare"
- Într-un sfârşit, fiecare excepţie ar trebui interceptată de un bloc **`catch`** din vreo metodă care nu numai declară într-o clauză **`throws`** ci şi interceptează clasa de excepţie respectivă

Regula "prinde sau declară"

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

Excepţii verificate şi neverificate

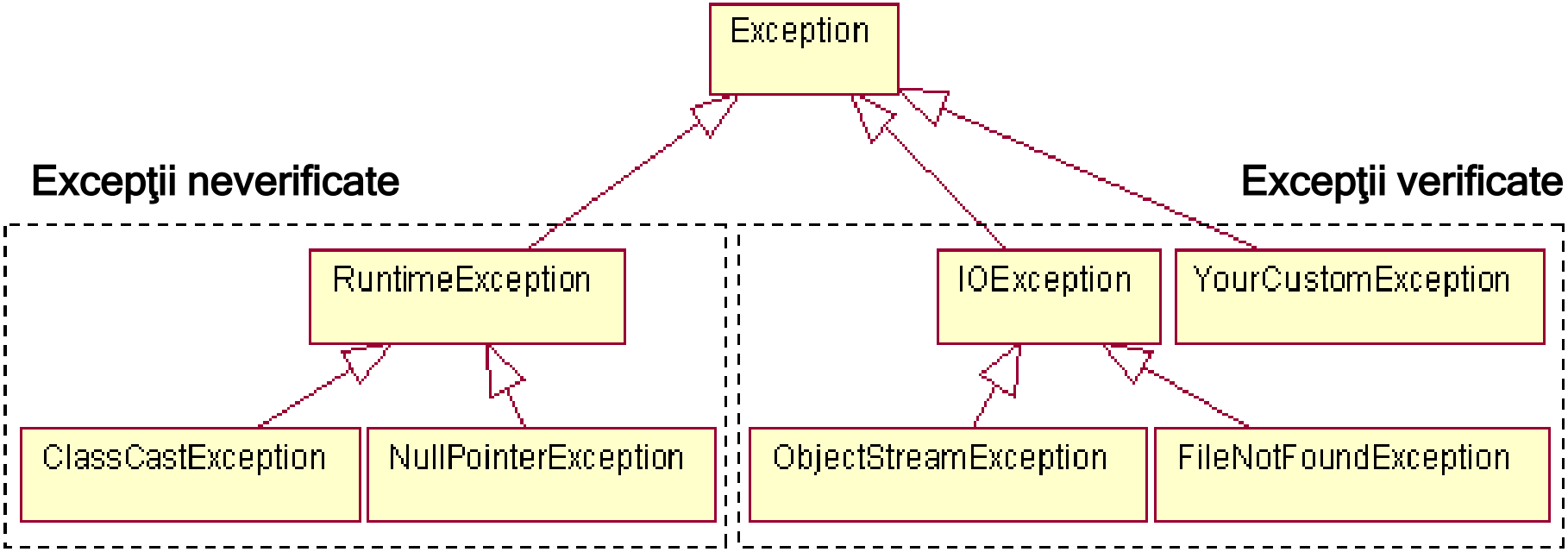
- Excepţiile care sunt supuse regulii "prinde sau declară" sunt numite **excepţii *verificate***
  - Compilatorul verifică pentru a vedea dacă excepţiile sunt luate în considerare fie într-un bloc `catch`, fie într-o clauză `throws`
  - Clasele `Throwable`, `Exception`, precum şi toţi descendenţii clasei `Exception` (excepție `RuntimeException`) constituie excepţii verificate

- **Toate celelalte excepţii sunt *neverificate***
  - Clasele `Error` și `RuntimeException` şi toate clasele care descind din ele constituie excepții *neverificate*
  - Aceste clase *nu sunt* supuse regulii "prinde sau declară"

Excepţii de la regula "prinde sau declară"

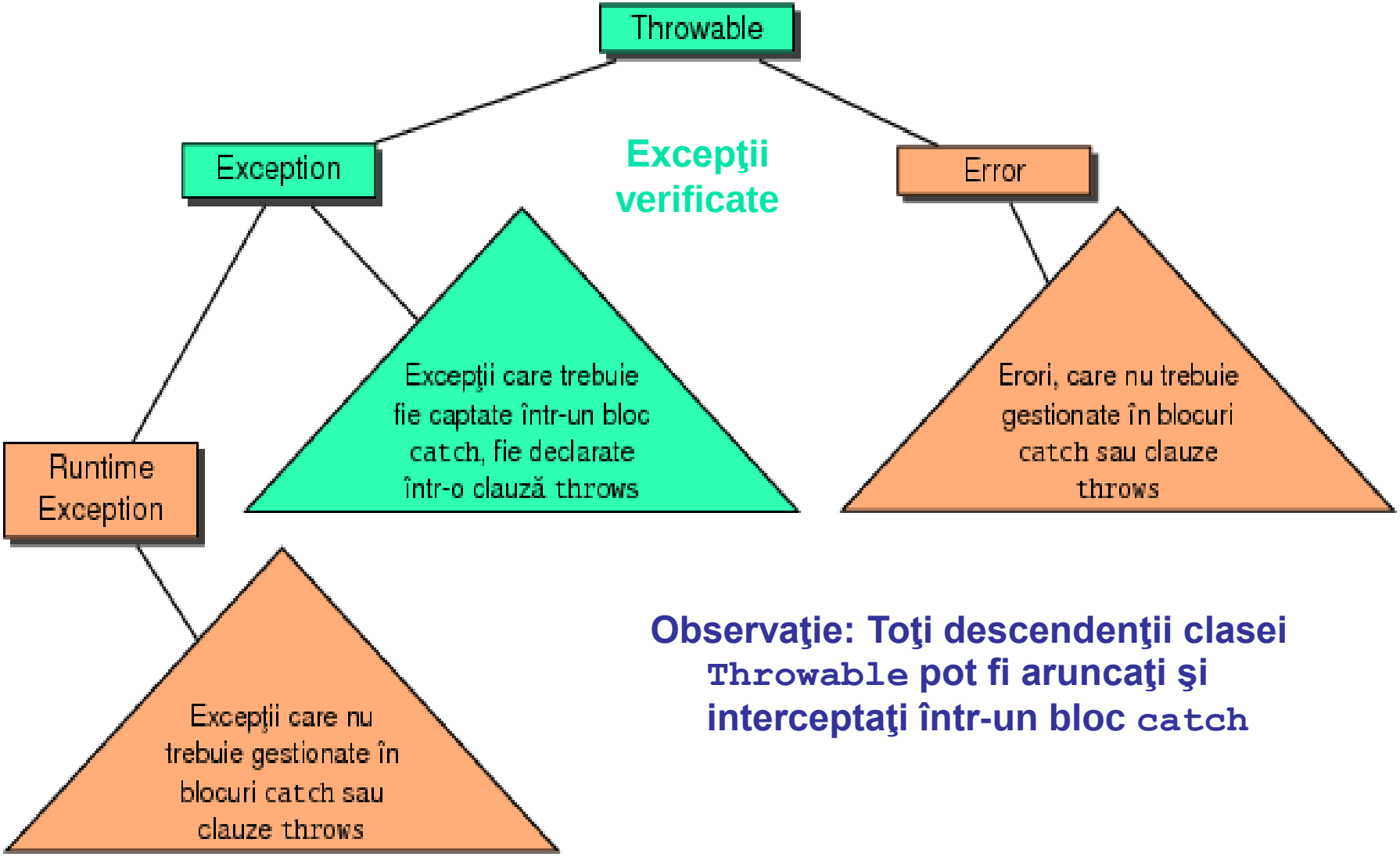
- Excepţiile *verificate* trebuie să respecte regula "prinde sau declară"
  - Programele în care pot fi aruncate aceste excepţii nu se vor compila până când excepţiile respective nu sunt tratate corespunzător
- Excepţiile *neverificate* nu sunt supuse regulii "prinde sau declară"
  - Programele în care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

Excepţii verificate şi neverificate



Notă. Aici este o mică parte a ierarhiei!

Ierarhia obiectelor Throwable (aruncabile)



Observaţie: Toţi descendenţii clasei `Throwable` pot fi aruncaţi şi interceptaţi într-un bloc `catch`

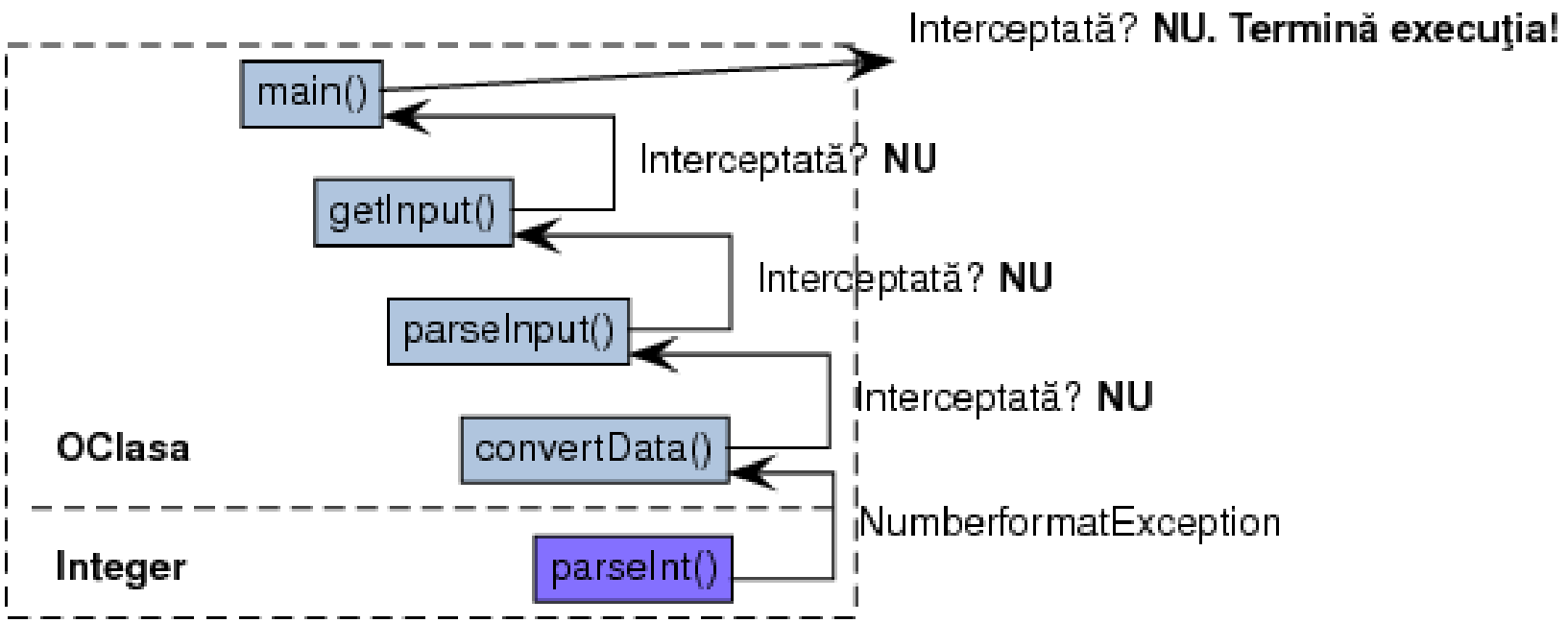
Clauza `throws` în clase derivate

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

Ce se întâmplă dacă o excepţie nu este interceptată?

- Dacă fiecare metodă până la, şi inclusiv, metoda **`main`** include o clauză **`throws`**, excepţia respectivă poate fi aruncată, dar poate să nu fie interceptată niciodată
  - Într-un program GUI (adică un program cu o interfaţa cu ferestre, grafică) nu se întâmplă nimic – atâta doar că utilizatorul poate fi lăsat într-o situaţie 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 excepţie
- Fiecare program bine scris trebuie în cele din urmă să intercepteze fiecare excepţie printr-un bloc **`catch`** în una dintre metodele sale

Propagarea excepţiei

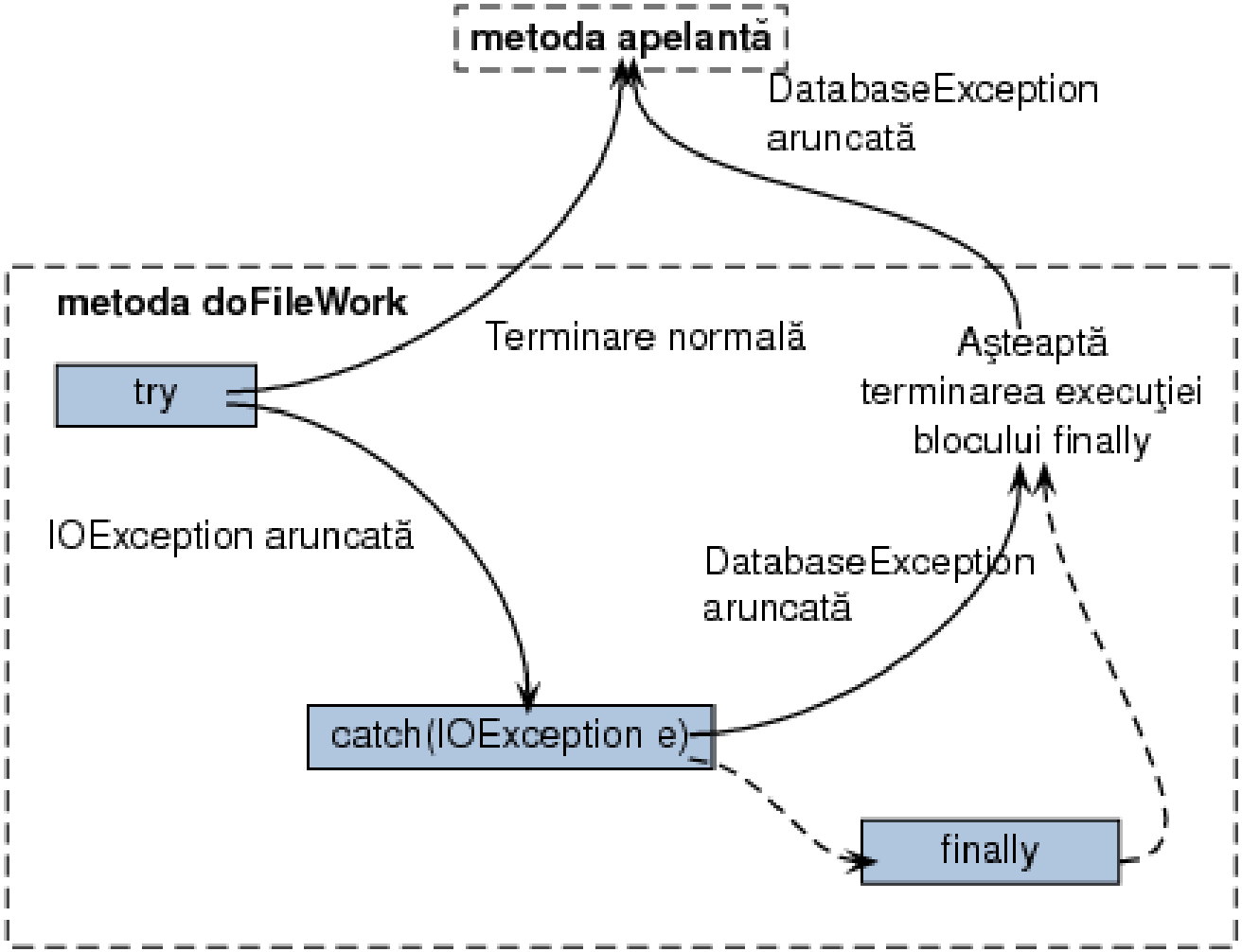


Un alt exemplu

```
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());
      }
   }
}
```

Explicaţii pentru exemplu



Când să folosim excepţiile

- Excepţiile trebuie rezervate pentru situaţiile în care o metodă întâlneşte *un caz neobişnuit sau neaşteptat, care nu poate fi tratat cu uşurinţă în vreun alt mod*

- Atunci când trebuie folosită tratarea excepţiilor, folosiţi aceste recomandări:
  - Includeţi instrucţiuni **`throw`** şi precizaţi clasele excepţie într-o clauză **`throws`** din definiţia metodei respective
  - Plasaţi blocurile **`try`** şi **`catch`** într-o metodă diferită

Când să folosim excepţiile

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

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

- Atunci când **oMetoda** este folosită de **altaMetoda**, **altaMetoda** trebuie să trateze excepţia:

```
public void altaMetoda()
{
  try {
    oMetoda();
    . . .
  }
  catch (OExceptie e) {
    CodPentruTratareaExceptiei
  }
  . . .
}
```

Ghid pentru excepţii

- Dacă metoda întâlneşte o condiţie anormală pe care nu o poate trata, atunci trebuie să arunce o excepţie
- Evitaţi folosirea excepţiilor pentru a indica situaţii care pot fi aşteptate ca parte a funcţionării normale a metodei
- Dacă metoda descoperă că clientul şi-a încălcat obligaţiile contractuale (spre exemplu, prin transmiterea de date de intrare neconforme specificaţiei), atunci aruncaţi o excepţie neverificată

Ghid pentru excepţii

- Dacă metoda nu-şi poate îndeplini contractul, atunci aruncaţi fie o excepţie verificată, fie una neverificată
- Dacă aruncaţi o excepţie pentru o situaţie anormală despre care consideraţi că programatorii trebuie să decidă în mod conştient cum să o trateze, atunci aruncaţi o excepţie verificată
- Definiţi sau alegeţi o clasă excepţie care există deja pentru fiecare fel de condiţie anormală care poate face ca metoda dvs. să arunce o excepţie

Re-aruncarea excepţiilor

- După interceptarea unei excepţii, ea poate fi re-aruncată dacă e cazul
- La re-aruncarea unei excepţii putem alege locaţia din care se va vedea ca aruncat obiectul în trasarea stivei de execuţie
  - Putem face ca excepţia re-aruncată să pară a fi aruncată din locul excepţiei originale sau din locul re-aruncării
- Pentru a re-arunca o excepţie cu indicarea locaţiei originale, pur şi simplu o aruncăm din nou:

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

Re-aruncarea excepţiilor

- Pentru a avea locaţia reală din care a fost re-aruncată apelăm metoda **`fillInStackTrace()`** a excepţiei
  - Metoda setează informaţia din trasarea stivei pe baza contextului de execuţie curent. Exemplu:

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

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

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)
```