

Laboratorul 12: Excepții

Video introductiv: [link](#)

Obiective

- înțelegerea conceptului de **excepție** și utilizarea corectă a mecanismelor de **generare și tratare a excepțiilor** puse la dispoziție de limbajul / mașina virtuală Java

Introducere

În esență, o **excepție** este un **eveniment** care se produce în timpul execuției unui program și care **perturbă** fluxul normal al instrucțiunilor acestuia.

De exemplu, în cadrul unui program care copiază un fișier, astfel de evenimente excepționale pot fi:

- absența fișierului pe care vrem să-l copiem
- imposibilitatea de a-l citi din cauza permisiunilor insuficiente
- probleme cauzate de accesul concurrent la fișier

Utilitatea conceptului de excepție

O abordare foarte des întâlnită, ce precedă apariția conceptului de excepție, este întoarcerea unor valori **speciale** din funcții care să desemneze situația apărută. De exemplu, în C, funcția `fopen` întoarce `NULL` dacă deschiderea fișierului a eșuat. Această abordare are două **dezavantaje** principale:

- câteodată, **toate** valorile tipului de retur ale funcției pot constitui rezultate valide. De exemplu, dacă definim o funcție care întoarce succesul unui număr întreg, nu putem întoarce o valoare specială în cazul în care se depășește valoarea maximă reprezentabilă (`Integer.MAX_VALUE`). O valoare specială, să zicem `-1`, ar putea fi interpretată ca numărul întreg `-1`.
- **nu** se poate **separa** secvența de instrucțiuni corespunzătoare execuției **normale** a programului de secvențele care tratează **erorile**. Firesc ar fi ca fiecare apel de funcție să fie urmat de verificarea rezultatului întors, pentru tratarea corespunzătoare a posibilelor erori. Această modalitate poate conduce la un cod foarte imbricat și greu de citit, de forma:

```
int openResult = open();

if (openResult == FILE_NOT_FOUND) {
    // handle error
} else if (openResult == INSUFFICIENT_PERMISSIONS) {
    // handle error
} else { // SUCCESS
    int readResult = read();
    if (readResult == DISK_ERROR) {
        // handle error
    } else {
        // SUCCESS
        ...
    }
}
```

Mecanismul bazat pe excepții înlătură ambele neajunsuri menționate mai sus. Codul ar arăta așa:

```
try {
    open();
    read();
    ...
} catch (FILE_NOT_FOUND) {
    // handle error
} catch (INSUFFICIENT_PERMISSIONS) {
    // handle error
} catch (DISK_ERROR) {
    // handle error
}
```

Se observă includerea instrucțiunilor ce aparțin fluxului normal de execuție într-un bloc **try** și precizarea condițiilor excepționale posibile la sfârșit, în câte un bloc **catch**. **Logica** este următoarea: se execută instrucțiune cu instrucțiune secvența din blocul try și, la apariția unei situații excepționale semnalate de o instrucțiune, **se abandonează** restul instrucțiunilor rămase neexecutate și **se sare** direct la blocul catch corespunzător.

Excepții în Java

Când o eroare se produce într-o funcție, aceasta creează un **obiect excepție** și îl pasează către `runtime system`. Un astfel de obiect conține informații despre situația apărută:

- **tipul** de excepție
- **stiva de apeluri** (stack trace): punctul din program unde a intervenit excepția, reprezentat sub forma lanțului de metode în care programul se află în acel moment

Pasarea menționată mai sus poartă numele de **aruncarea** (throwing) unei excepții.

Aruncarea excepțiilor

Exemplu de **aruncare** a unei excepții:

```
List<String> l = getArrayListObject();
if (null == l)
    throw new Exception("The list is empty");
```

În acest exemplu, încercăm să obținem un obiect de tip `ArrayList`; dacă funcția `getArrayListObject` întoarce `null`, aruncăm o excepție.

Pe exemplul de mai sus putem face următoarele observații:

- un **obiect-excepție** este un obiect ca oricare altul, și se instanțiază la fel (folosind `new`);
- aruncarea excepției se face folosind cuvântul cheie **throw**;
- există clasa `Exception` care desemnează comportamentul specific pentru excepții.

În realitate, clasa `Exception` este părintele majorității claselor excepție din Java. Enumerăm câteva excepții standard:

- `IndexOutOfBoundsException`: este aruncată când un index asociat unei liste sau unui vector depășește dimensiunea colecției respective.
- `NullPointerException`: este aruncată când se accesează un obiect neinstanțiat (`null`).

- **NoSuchElementException**: este aruncată când se apelează `next` pe un `Iterator` care nu mai conține un element următor.

În momentul în care se instanțiază un obiect-excepție, în acesta se reține întregul lanț de apeluri de funcții prin care s-a ajuns la instrucțiunea curentă. Această succesiune se numește **stack trace** și se poate afișa prin apelul `e.printStackTrace()`, unde `e` este obiectul excepție.

Prinderea excepțiilor

Când o excepție a fost aruncată, `runtime system` încearcă să o trateze (**prindă**). Tratarea unei excepții este făcută de o porțiune de cod **specială**.

- Cum definim o astfel de porțiune de cod **specială**?
- Cum specificăm faptul că o porțiune de cod specială tratează o **anumită** excepție?

Să observăm următorul exemplu:

```
public void f() throws Exception {
    List<String> l = null;

    if (null == l)
        throw new Exception();
}

public void catchFunction() {
    try {
        f();
    } catch (Exception e) {
        System.out.println("Exception found!");
    }
}
```

Se observă că dacă o funcție aruncă o excepție și **nu** o prinde trebuie, în general, să adauge **clauza throws** în antet.

Funcția `f` va arunca întotdeauna o excepție (din cauza că `l` este mereu `null`). Observați cu atenție funcția `catchFunction`:

- în interiorul său a fost definit un bloc `try`, în interiorul căruia se apelează `f`. De obicei, pentru a **prinde** o excepție, trebuie să specificăm o zonă în care așteptăm ca excepția să se producă (**guarded region**). Această zonă este introdusă prin `try`.
- în continuare, avem blocul `catch (Exception e)`. La producerea excepției, blocul `catch` corespunzător va fi executat. În cazul nostru se va afișa mesajul "Exception found!". După aceea, programul va continua să ruleze normal în continuare.

Observați un alt exemplu:

```
public void f() throws NullPointerException, EmptyListException {
    List<String> l = generateList();

    if (l == null)
        throw new NullPointerException();
}
```

```
    if (l.isEmpty())
        throw new EmptyListException();
}

public void catchFunction() {
    try {
        f();
    } catch (NullPointerException e) {
        System.out.println("Null Pointer Exception found!");
    } catch (EmptyListException e) {
        System.out.println("Empty List Exception found!");
    }
}
```

În acest exemplu funcția `f` a fost modificată astfel încât să existe posibilitatea de a arunca `NullPointerException` sau `EmptyListException`. Observați faptul că în `catchFunction` avem două blocuri `catch`. În funcție de excepția aruncată de `f`, numai un singur bloc `catch` se va executa.

Prin urmare:

- putem defini mai multe blocuri `catch` pentru a implementa o tratare preferențială a excepțiilor, în funcție de tipul acestora
- în cazul aruncării unei excepții într-un bloc `try`, se va intra **într-un singur bloc** `catch` (cel aferent excepției aruncate)

Nivelul la care o excepție este tratată depinde de logica aplicației. Acesta **nu** trebuie să fie neapărat nivelul imediat următor ce invocă secțiunea generatoare de excepții. Desigur, propagarea de-a lungul mai multor nivele (metode) presupune utilizarea clauzei `throws`.

Dacă o excepție nu este tratată nici în `main`, aceasta va conduce la **încheierea** execuției programului!

Blocuri `try-catch` imbricate

În general, vom dispune în același bloc `try-catch` instrucțiunile care pot fi privite ca înfăptuind un același scop. Astfel, dacă o operație din secvență eșuează, se renunță la instrucțiunile rămase și se sare la un bloc `catch`.

Putem specifica operații opționale, al căror eșec să **nu influențeze** întreaga secvență. Pentru aceasta folosim blocuri `try-catch` **imbricate**:

```
try {
    op1();

    try {
        op2();
        op3();
    } catch (Exception e) { ... }

    op4();
}
```

```
    op5();  
} catch (Exception e) { ... }
```

Dacă apelul `op2` eșuează, se renunță la apelul `op3`, se execută blocul `catch` interior, după care se continuă cu apelul `op4`.

Blocul `finally`

Presupunem că în secvența de mai sus, care deschide și citește un fișier, avem nevoie să închidem fișierul deschis, atât în cazul normal, cât și în eventualitatea apariției unei erori. În aceste condiții se poate atașa un bloc `finally` după ultimul bloc `catch`, care se va executa în **ambele** cazuri menționate.

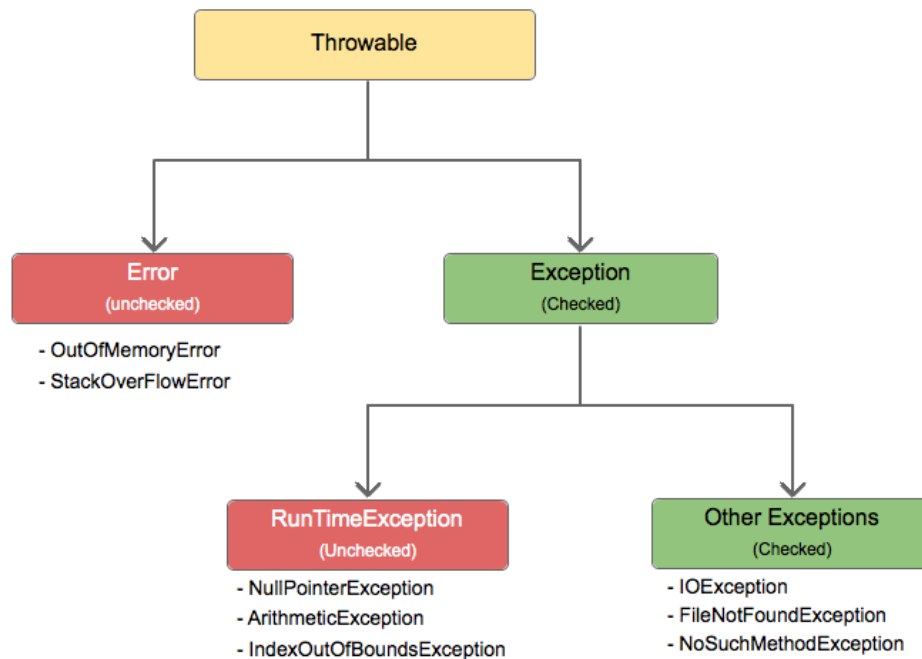
Secvența de cod următoare conține o structură `try-catch-finally`:

```
try {  
    open();  
    read();  
    ...  
} catch (FILE_NOT_FOUND) {  
    // handle error  
} catch (INUFFICIENT_PERMISSIONS) {  
    // handle error  
} catch (DISK_ERROR) {  
    // handle error  
} finally {  
    // close file  
}
```

Blocul `finally` se dovedește foarte util când în blocurile `try-catch` se găsesc instrucțiuni **return**. El se va executa și în **acest** caz, exact înainte de execuția instrucțiunii **return**, aceasta fiind executată ulterior.

Tipuri de excepții

Nu toate excepțiile trebuie prinse cu `try-catch`. Pentru a înțelege de ce, să analizăm clasificarea excepțiilor:



Clasa **Throwable**:

- Superclasa tuturor erorilor și excepțiilor din Java.
- Doar obiectele ce extind această clasă pot fi aruncate de către JVM sau prin instrucțiunea `throw`.
- Numai această clasă sau una dintre subclasele sale pot fi tipul de argument într-o clauză `catch`.

Checked exceptions, ce corespund clasei **Exception**:

- O aplicație bine scrisă ar trebui să le **prindă** și să permită **continuarea** rulării programului.
- Să luăm ca exemplu un program care cere utilizatorului un nume de fișier (pentru a-l deschide). În mod normal, utilizatorul va introduce un nume de fișier care există și care poate fi deschis. Există însă posibilitatea ca utilizatorul să greșescă, caz în care se va arunca o excepție `FileNotFoundException`.
- Un program bine scris va prinde această excepție, va afișa utilizatorului un mesaj de eroare, și îi va permite eventual să reintroducă un nou nume de fișier.

Errors, ce corespund clasei **Error**:

- Acestea definesc situații excepționale declanșate de factori **externi** aplicației, pe care aceasta nu le poate anticipa și nu-și poate reveni, dacă se produc.
- Spre exemplu, alocarea unui obiect foarte mare (un vector cu milioane de elemente), poate arunca `OutOfMemoryError`.
- Aplicația poate încerca să prindă această eroare, pentru a anunța utilizatorul despre problema apărută; după aceasta însă, programul va eșua (afișând eventual `stack trace`).

Runtime Exceptions, ce corespund clasei **RuntimeException**:

- Ca și erorile, acestea sunt condiții excepționale, însă, spre **deosebire** de **erori**, ele sunt declanșate de factori **interni** aplicației. Aplicația nu poate anticipa și nu își poate reveni dacă acestea sunt aruncate.
- **Runtime exceptions** sunt produse de diverse bug-uri de programare (erori de logică în aplicație, folosire necorespunzătoare a unui API, etc).
- Spre exemplu, a realiza apeluri de metode sau membri pe un obiect `null` va produce

`NullPointerException`. Firește, putem prinde excepția. Mai **natural** însă ar fi să **eliminăm** din program un astfel de bug care ar produce excepția.

Excepțiile **checked** sunt cele **prinse** de blocurile `try-catch`. Toate excepțiile sunt **checked**, mai puțin cele de tip **Error**, **RuntimeException** și subclasele acestora, adică cele de tip **unchecked**.

Nu este indicată prinderea excepțiilor **unchecked** (de tip `Error` sau `RuntimeException`) cu `try-catch`.

Putem arunca `RuntimeException` fără să o menționăm în clauza `throws` din antet:

```
public void f(Object o) {
    if (o == null)
        throw new NullPointerException("o is null");
}
```

Definirea de excepții noi

Când aveți o situație în care alegerea unei excepții (de aruncat) nu este evidentă, puteți opta pentru a scrie propria voastră excepție, care să extindă `Exception`, `RuntimeException` sau `Error`.

Exemplu:

```
class TemperatureException extends Exception {}

class TooColdException extends TemperatureException {}

class TooHotException extends TemperatureException {}
```

În aceste condiții, trebuie acordată atenție **ordinii** în care se vor defini blocurile `catch`. Acestea trebuie precizate de la clasa excepție cea mai **particulară**, până la cea mai **generală** (în sensul moștenirii). De exemplu, pentru a întrebuința excepțiile de mai sus, blocul `try-catch` ar trebui să arate ca mai jos:

```
try {
    ...
} catch (TooColdException e) {
    ...
} catch (TemperatureException e) {
    ...
} catch (Exception e) {
    ...
}
```

Afirmația de mai sus este motivată de faptul că întotdeauna se alege **primul** bloc `catch` care se potrivește cu tipul excepției apărute. Un bloc `catch` referitor la o clasă excepție **părinte**, ca `TemperatureException` prinde și excepții de tipul claselor **copil**, ca `TooColdException`. Poziționarea unui bloc mai general **înaintea** unuia mai particular ar conduce la **ignorarea** blocului particular.

Din **Java 7** se pot prinde mai multe excepții în același catch. Sintaxa este:

```
try {  
    ...  
} catch(IOException | FileNotFoundException e) {  
    ...  
}
```

Din **Java 7**, a fost adăugată construcția try-with-resources, care ne permite să declarăm resursele într-un bloc try, cu asigurarea că resursele vor fi închise după executarea acelui bloc. Resursele declarate trebuie să implementeze interfața [AutoCloseable](#).

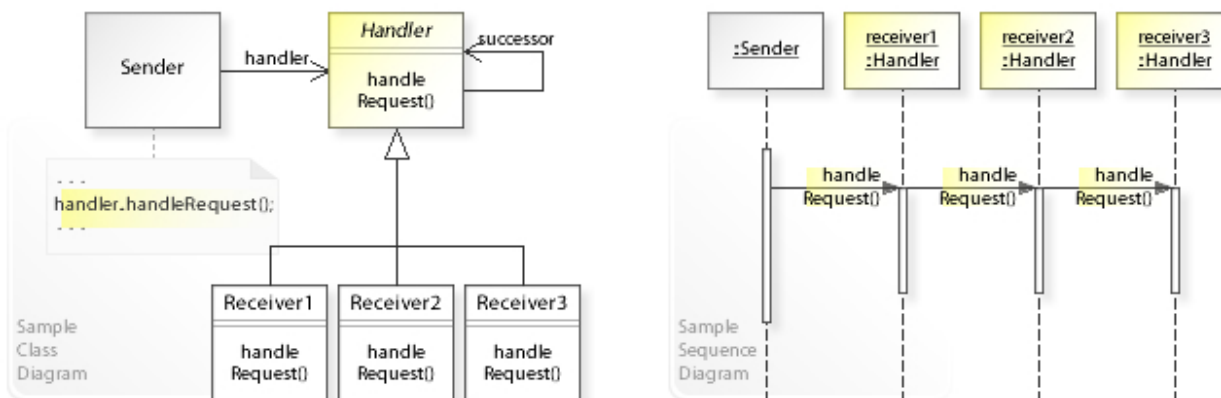
```
try (PrintWriter writer = new PrintWriter(file)) {  
    writer.println("Hello World");  
}
```

Excepțiile în contextul moștenirii

Metodele suprascrise (overriden) pot arunca **numai** excepțiile specificate de metoda din **clasa de bază** sau excepții **derivate** din acestea.

Chain-of-responsibility Pattern

În proiectarea orientată pe obiect, pattern-ul "Chain-of-responsibility" (lanț de responsabilitate) este un model de design constând dintr-o sursă de obiecte de comandă și o serie de obiecte de procesare. Fiecare obiect de procesare conține logică care definește tipurile de obiecte de comandă pe care le poate gestiona; restul sunt transferate către următorul obiect de procesare din lanț. De asemenea, există un mecanism pentru adăugarea de noi obiecte de procesare la sfârșitul acestui lanț. Astfel, lanțul de responsabilitate este o versiune orientată pe obiecte a `if ... else if ... else if else ... endif`, cu avantajul că blocurile condiție-acțiune pot fi dinamic rearanjate și reconfigurate la timpul de execuție.



Într-o variantă a modelului standard al lanțului de responsabilitate, un handler poate acționa ca un [dispatcher](#), capabil să trimită comenzi în diverse direcții, formând un arbore de responsabilități (tree

of responsibility). În unele cazuri, acest lucru poate apărea recursiv, cu procesarea obiectelor care apelează obiecte de procesare de nivel superior cu comenzi care încearcă să rezolve o parte mai mică a problemei; în acest caz, recurența continuă până când comanda este procesată, sau întregul arbore a fost explorat. Un interpretor XML ar putea funcționa în acest mod.

Exerciții

- (4p)** Definiți o clasă care să implementeze operații pe numere **double**. Operațiile vor arunca excepții. Clasa va trebui să implementeze interfața `CalculatorBase`, ce conține trei metode:
 - `add`: primește două numere și întoarce un `double`
 - `divide`: primește două numere și întoarce un `double`
 - `average`: primește o colecție ce conține obiecte `double`, și întoarce media acestora ca un număr de tip `double`. !! Pentru calculul mediei, sunt folosite metodele `add` și `divide` !! .
 - Metodele pot arunca următoarele excepții (definite în interfața `Calculator`):
 - `NullPointerException`: este aruncată dacă vreunul din parametrii primiți este `null`;
 - `OverflowException`: este aruncată dacă suma a două numere e egală cu `Double.POSITIVE_INFINITY`;
 - `UnderflowException`: este aruncată dacă suma a două numere e egală cu `Double.NEGATIVE_INFINITY`.
 - Completați metoda `main` din clasa `MainEx2`, evidențiind prin teste toate cazurile posibile care generează excepții.
- (4p)** Vom realiza o mini librărie online în care putem adăuga cărți cumpărându-le și din care putem să extragem o carte deja existentă.
 1. Definește clasa `Book` care are parametrii `title`, `author`, `genre` și `price`.
 2. Definește două noi excepții care extind clasa `Exception`:
 - `NotEnoughMoneyException`, care e aruncată atunci când utilizatorul nu are bani suficienți pentru a cumpăra o carte
 - `NoSuchBookException`, care e aruncată atunci când cartea dorită nu se găsește în librărie.
 3. Definește clasa `OnlineLibrary` care are:
 - doi parametri: o listă de cărți și bugetul utilizatorului
 - un constructor care primește bugetul inițial al utilizatorului
 - metodele:
 - `addBook` - primește o carte și o adaugă în librărie dacă utilizatorul are fonduri suficiente
 - `getBook` - returnează cartea dorită, dacă aceasta se află în librărie.
 4. În metoda `Main` să se realizeze TODO-urile:
 - `TODO1` - adaugă lista de cărți în librărie
 - `TODO2` - ia cartea `book4` din librărie. Dacă nu există, adaug-o.
 - Atenție la tratarea excepțiilor! (A se afișa un mesaj corespunzător fiecărui caz, ca în exemplu).
- (2p)** Dorim să implementăm un `Logger` pe baza pattern-ului `Chain-of-responsibility`, definit în laborator, pe care îl vom folosi să păstrăm un jurnal de evenimente al unui program:
 1. Creați enumerația `LogLevel`, ce va acționa ca un bitwise flag, care va conține:
 - valorile - `Info`, `Debug`, `Warning`, `Error`, `FunctionalMessage`, `FunctionalError`.
 - Această enumerație va expune și o metodă statică `all()` care va întoarce o colecție de `EnumSet<LogLevel>` în care vor fi toate valorile de mai sus (Hint: `EnumSet.allOf()`).
 2. Creați o clasă abstractă `LoggerBase` care:
 - va primi în constructor un obiect de tip `EnumSet<LogLevel>` care va defini pentru ce nivele de log se va afișa mesajul
 - va păstra o referință către următorul `LoggerBase` la care se trimite mesajul
 - va expune o metodă publică `setNext` ce va primi un `LoggerBase` și va seta următorul delegat din lista de responsabilitate
 - va defini o metodă abstractă `protected writeMessage` ce va primi mesajul care trebuie afișat

și afișează mesajul în cauză

- va expune o metodă publică `message` ce va primi mesajul care trebuie afișat și o severitate de tip `LogLevel` (adică `Info`, `Debug`, `Warning`, `Error`, `FunctionalMessage` sau `FunctionalError`). Dacă instanța de logger conține această severitate în colecția primită în constructor, atunci se va apela metoda `writeMessage`. Apoi se vor pasa mesajul și severitatea către următorul delegat din lista de responsabilitate (dacă există unul)
3. Definiți clasele de mai jos care vor extinde `LoggerBase` și implementa metoda `writeMessage`:
- `ConsoleLogger` - care va scrie toate tipurile de `LogLevel` (Hint: `all()`) și va prefixa mesajele cu `[Console]`
 - `EmailLogger` - care va scrie doar tipurile `FunctionalMessage` și `FunctionalError` și va prefixa mesajele cu `[Email]`
 - `FileLogger` - care va scrie doar tipurile `Warning` și `Error` și va prefixa mesajele cu `[File]`
4. Completați cele 2 TODO-uri rămase în metoda `main` din clasa `Main`. (Hint: `EnumSet.of()` pentru constructori)

Referințe

- [Exception](#)
- [Error](#)
- [RuntimeException](#)
- [NullPointerException](#)
- [IndexOutOfBoundsException](#)
- [NoSuchElementException](#)
- [OutOfMemoryError](#)
- [StackOverflowError](#)
- [Adaptarea exercițiului după exemplul din C#](#)

From:
<http://ocw.cs.pub.ro/courses/> - **CS Open CourseWare**

Permanent link:
<http://ocw.cs.pub.ro/courses/poo-ca-cd/laboratoare/exceptii>



Last update: **2024/01/14 17:22**