Laboratorul 12: Excepții

Video introductiv: link

Obiective

- înţelegerea conceptului de **excepţie** şi utilizarea corectă a mecanismelor de **generare şi tratare** a excepţiilor puse la dispoziţie de limbajul / maşina virtuală Java

Introducere

În esenţă, o **excepţie** este un **eveniment** care se produce în timpul execuţiei unui program şi care **perturbă** fluxul normal al instrucţiunilor acestuia.

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

- absenţa fişierului pe care vrem să-l copiem
- imposibilitatea de a-l citi din cauza permisiunilor insuficiente
- probleme cauzate de accesul concurent la fişier

Utilitatea conceptului de excepţie

O abordare foarte des intâlnită, ce precedă apariţia conceptului de excepţie, este întoarcerea unor valori **speciale** din funcţii care să desemneze situaţia apărută. De exemplu, în C, funcţia fopen întoarce NULL dacă deschiderea fişierului a eşuat. Această abordare are două **dezavantaje** principale:

- câteodată, **toate** valorile tipului de retur ale funcţiei pot constitui rezultate valide. De exemplu, dacă definim o funcţie care întoarce succesorul unui numar întreg, nu putem întoarce o valoare specială în cazul în care se depăşeşte valoarea maximă reprezentabilă (Integer.MAX_VALUE). O valoare specială, să zicem -1, ar putea fi interpretată ca numărul întreg -1.
- **nu** se poate **separa** secvenţa de instrucţiuni corespunzătoare execuţiei **normale** a programului de secvenţele care trateaza **erorile**. Firesc ar fi ca fiecare apel de funcţie să fie urmat de verificarea rezultatului întors, pentru tratarea corespunzătoare a posibilelor erori. Această modalitate poate conduce la un cod foarte imbricat şi 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 excepţii înlătură ambele neajunsuri menţionate mai sus. Codul ar arăta aşa:

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

Se observă includerea instrucţiunilor ce aparţin fluxului normal de execuţie într-un bloc **try** şi precizarea condiţiilor excepţionale posibile la sfârşit, în câte un bloc **catch**. **Logica** este următoarea: se execută instrucţiune cu instrucţiune secvenţa din blocul try şi, la apariţia unei situaţii excepţionale semnalate de o instrucţiune, **se abandonează** restul instrucţiunilor rămase neexecutate şi **se sare** direct la blocul catch corespunzător.

Excepţii în Java

Când o eroare se produce într-o funcţie, aceasta creează un **obiect excepţie** şi îl pasează către `runtime system`. Un astfel de obiect conţine informaţii despre situaţia apărută:

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

Pasarea menţionată mai sus poartă numele de **aruncarea** (throwing) unei excepţii.

Aruncarea excepţiilor

Exemplu de **aruncare** a unei excepţii:

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

În acest exemplu, încercăm să obţinem un obiect de tip `ArrayList`; dacă funcţia `getArrayListObject` întoarce null, aruncăm o excepţie.

Pe exemplul de mai sus putem face următoarele observaţii:

- un **obiect-excepţie** este un obiect ca oricare altul, şi se instanţiază la fel (folosind new);
- aruncarea excepţiei se face folosind cuvântul cheie **throw**;
- există clasa Exception care desemnează comportamentul specific pentru excepţii.

În realitate, clasa `Exception` este părintele majorităţii claselor excepţie din Java. Enumerăm câteva excepţii standard:

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

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

În momentul în care se instanţiază un obiect-excepţie, în acesta se reţine întregul lanţ de apeluri de funcţii prin care s-a ajuns la instrucţiunea curentă. Această succesiune se numeşte **stack trace** şi se poate afişa prin apelul e.printStackTrace(), unde e este obiectul excepţie.

Prinderea excepţiilor

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

- Cum definim o astfel de porţiune de cod **specială**?
- Cum specificăm faptul că o porţiune de cod specială tratează o **anumită** excepţie?

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 funcţie aruncă o excepţie şi **nu** o prinde trebuie, în general, să adauge **clauza throws** în antet.

Funcţia `f` va arunca întotdeauna o excepţie (din cauza că `l` este mereu `null`). Observaţi cu atenţie funcţia `catchFunction`:

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

Observaţi 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 funcţia `f` a fost modificată astfel încât să existe posibilitatea de a arunca `NullPointerException` sau `EmptyListException`. Observaţi 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 preferenţială a excepţiilor, în funcţie 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 excepţie este tratată depinde de logica aplicaţiei. Acesta **nu** trebuie să fie neapărat nivelul imediat următor ce invocă secţiunea generatoare de excepţii. Desigur, propagarea de-a lungul mai multor nivele (metode) presupune utilizarea clauzei `throws`.

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

Blocuri try-catch imbricate

În general, vom dispune în acelaşi bloc `try-catch` instrucţiunile care pot fi privite ca înfăptuind un acelaşi scop. Astfel, dacă **o** operaţie din secvență eșuează, se renunţă la instrucţiunile rămase şi se sare la un bloc `catch`.

Putem specifica operaţii opţionale, al căror eşec să **nu influenţeze** întreaga secvenţă. Pentru aceasta folosim blocuri `try-catch` **imbricate**:

```
try {
    op1();

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

    op4();
```

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

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

Blocul finally

Presupunem că în secvenţa de mai sus, care deschide şi citeşte un fişier, avem nevoie să închidem fişierul deschis, atât în cazul normal, cât şi în eventualitatea apariţiei unei erori. În aceste condiţii se poate ataşa un bloc `finally` după ultimul bloc `catch`, care se va executa în **ambele** cazuri menţionate.

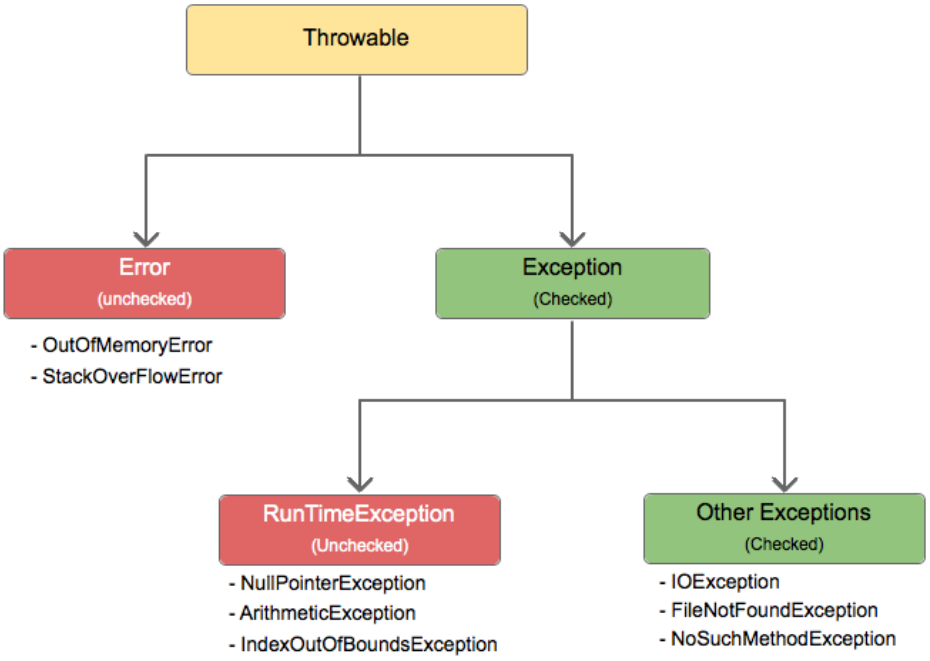
Secvenţa de cod următoare conţine o structură `try-catch-finally`:

```
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 dovedeşte foarte util când în blocurile `try-catch` se găsesc instrucţiuni **return**. El se va executa şi în **acest** caz, exact înainte de execuţia instrucţiunii **return**, aceasta fiind executată ulterior.

Tipuri de excepţii

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



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

Errors, ce corespund clasei Error:

- Acestea definesc situaţii excepţionale declanşate de factori **externi** aplicaţiei, pe care aceasta nu le poate anticipa şi nu-şi poate reveni, dacă se produc.
- Spre exemplu, alocarea unui obiect foarte mare (un vector cu milioane de elemente), poate arunca `OutOfMemoryError`.
- Aplicaţia poate încerca să prindă această eroare, pentru a anunţa utilizatorul despre problema apărută; după aceasta însă, programul va eşua (afişând eventual `stack trace`).

Runtime Exceptions, ce corespund clasei RuntimeException:

- Ca şi erorile, acestea sunt condiţii excepţionale, însă, spre **deosebire** de **erori**, ele sunt declanşate de factori **interni** aplicaţiei. Aplicaţia nu poate anticipa şi nu îşi poate reveni dacă acestea sunt aruncate.
- **Runtime exceptions** sunt produse de diverse bug-uri de programare (erori de logică în aplicaţie, folosire necorespunzătoare a unui API, etc).
- Spre exemplu, a realiza apeluri de metode sau membri pe un obiect `null` va produce

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

Excepţiile **checked** sunt cele **prinse** de blocurile `try-catch`. Toate excepţiile sunt **checked**, mai puțin cele de tip **Error**, **RuntimeException** şi subclasele acestora, adică cele de tip **unchecked**.

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

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

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

Definirea de excepţii noi

Când aveţi o situaţie în care alegerea unei excepţii (de aruncat) nu este evidentă, puteţi opta pentru a scrie propria voastră excepţie, care să extindă Exception, RuntimeException sau Error.

Exemplu:

```
class TemperatureException extends Exception {}

class TooColdException extends TemperatureException {}

class TooHotException extends TemperatureException {}
```

În aceste condiţii, trebuie acordată atenţie **ordinii** în care se vor defini blocurile catch. Acestea trebuie precizate de la clasa excepţie cea mai **particulară**, până la cea mai **generală** (în sensul moştenirii). De exemplu, pentru a întrebuinţa excepţiile de mai sus, blocul `try-catch` ar trebui să arate ca mai jos:

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

Afirmaţia de mai sus este motivată de faptul că întotdeauna se alege **primul** bloc catch care se potriveşte cu tipul excepţiei apărute. Un bloc catch referitor la o clasă excepţie **părinte**, ca `TemperatureException` prinde şi excepţii de tipul claselor **copil**, ca `TooColdException`. Poziţionarea unui bloc mai general **înaintea** unuia mai particular ar conduce la **ignorarea** blocului particular.

Din **Java 7** se pot prinde mai multe excepţii în acelaşi 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.

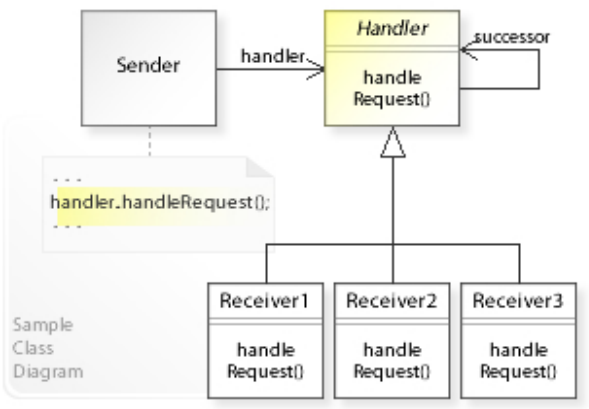
```
try (PrintWriter writer = new PrintWriter(file)) {
    writer.println("Hello World");
}
```

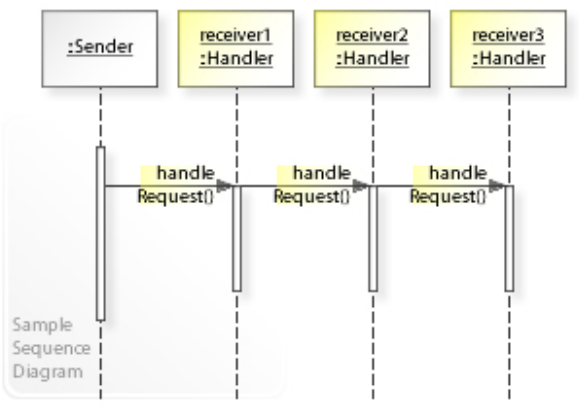
Excepţiile în contextul moştenirii

Metodele suprascrise (overriden) pot arunca **numai** excepţiile specificate de metoda din **clasa de bază** sau excepţii **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.

Exerciţii

1. **(4p)** Definiţi o clasă care să implementeze operaţii pe numere **double**. Operaţiile vor arunca excepţii. Clasa va trebui să implementeze interfața `CalculatorBase`, ce conţine trei metode:
   - `add`: primeşte două numere şi întoarce un `double`
   - `divide`: primeşte două numere şi întoarce un `double`
   - `average`: primeşte o colecţie ce conţine obiecte `double`, şi întoarce media acestora ca un numar de tip `double`. !! Pentru calculul mediei, sunt folosite metodele `add` şi `divide` !! .
   - Metodele pot arunca următoarele excepții (definite în interfața `Calculator`):
     - `NullParameterException`: 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 excepţii.
2. **(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 parametrii: 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).
3. **(2p)** Dorim să implementăm un `Logger` pe baza pattern-ului Chain-of-responsibility, definit în laborator, pe care îl vom folosi să păstram 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 afisa 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)

Referinţe

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