Exceptii

De la POO

Salt la: Navigare, căutare

Cuprins

[ascunde] [ascunde]

- 1 Introducere
  - 1.1 Ce este o exceptie
  - 1.2 Utilitatea conceptului de exceptie
- 2 Exceptii in Java
  - 2.1 Aruncarea exceptiilor
  - 2.2 Prinderea exceptiilor
  - 2.3 Blocuri try-catch imbricate
  - 2.4 Blocul finally
  - 2.5 Tipuri de exceptii
  - 2.6 Definirea de exceptii noi
  - 2.7 Exceptiile in contextul mostenirii
- 3 Exercitii
- 4 Referinte

Introducere

Ce este o exceptie

In esenta, o **exceptie** este un **eveniment** care se produce in timpul executiei unui program si care **perturba** fluxul normal al instructiunilor acestuia. De exemplu, in cadrul unui program care copiaza un fisier, astfel de evenimente exceptionale pot fi:

- absenta fisierului pe care vrem sa-l copiem
- imposibilitatea de a-l citi din cauza permisiunilor insuficiente sau din cauza unei zone invalide de pe hard-disk
- etc

Utilitatea conceptului de exceptie

O abordare foarte des intalnita, ce preceda aparitia conceptului de exceptie, este intoarcerea unor valori **speciale** din functii, care sa desemneze situatia aparuta. De exemplu, in `C`, functia `fopen` intoarce `NULL` daca deschiderea fisierului a esuat. Aceasta abordare are doua **dezavantaje** principale:

- cateodata, **toate** valorile tipului de retur al functiei pot constitui rezultate valide. De exemplu, daca definim o functie care intoarce succesorul unui numar intreg, nu putem intoarce o valoare speciala in cazul in care se depasesete valoarea maxima reprezentabila (`Integer.MAX_VALUE`). O valoare speciala, ca `-1`, ar putea fi interpretata ca numarul intreg `-1`.
- **nu** se poate **separa** secventa de instructiuni corespunzatoare executiei **normale** a programului de secventele care trateaza **erorile**. Firesc ar fi ca fiecare apel de functie sa fie urmat de verificarea rezultatului intors, pentru tratarea corespunzatoare a posibilelor erori. Aceasta modalitate poate conduce la un cod foarte imbricat si greu de citit, de forma:

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

Mecanismul bazat pe **exceptii** inlatura ambele neajunsuri mentionate mai sus. Codul ar arata asa:

```
try {
    open();
```

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

Se observa includerea instructiunilor ce apartin fluxului normal de executie intr-un bloc **`try`**, si precizarea conditiilor exceptionale posibile la sfarsit, in cate un bloc **`catch`**. **Logica** este urmatoarea: se executa instructiune cu instructiune secventa din blocul `try` si, la aparitia unei situatii exceptionale, semnalate de o instructiune, **se abandoneaza** restul instructiunilor ramase neexecutate si **se sare** direct la blocul `catch` corespunzator.

Exceptii in Java

Cand o eroare se produce intr-o functie, aceasta creeaza un **obiect exceptie** si il *paseaza* catre *runtime system*. Un astfel de obiect contine informatii despre situatia aparuta:

- **tipul** exceptiei
- **stiva de apeluri** (*stack trace*): punctul din program unde a intervenit exceptia, reprezentat sub forma lantului de metode (obtinut prin invocarea succesiva a metodelor din alte metode) in care programul se afla in acel moment.

Pasarea mentionata mai sus poarta numele de **aruncarea** (*throwing*) unei exceptii.

Aruncarea exceptiilor

Exemplu de **aruncare** a unei exceptii:

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

In acest exemplu, incercam sa obtinem un obiect de tip `ArrayList`; daca functia `getArrayListObject` intoarce `null`, aruncam o exceptie.

Pe exemplul de mai sus putem face urmatoarele observatii:

- un **obiect-exceptie** este un obiect ca oricare altul, si se instantiaza la fel (folosind `new`)
- aruncarea exceptiei se face folosind cuvantul cheie **`throw`**
- exista clasa **`Exception`** care desemneaza comportamentul specific pentru exceptii.

In realitate, clasa `Exception` este parintele majoritatii claselor exceptie din Java. Enumeram cateva exceptii standard:

- **`IndexOutOfBoundsException`**: Este aruncata cand un index asociat unei liste sau unui vector depaseste dimensiunea colectiei respective.
- **`NullPointerException`**: Este aruncata cand se acceseaza un obiect neinstantiat (`null`).
- **`NoSuchElementException`**: Este aruncata cand se apeleaza `next` pe un `Iterator` care nu mai contine un element urmator.

In momentul in care se instantiaza un obiect exceptie, in acesta se retine intregul lant de apeluri de functie, prin care s-a ajuns la instructiunea curenta. Aceasta succesiune se numeste ***stack trace***, si se poate afisa prin apelul `e.printStackTrace()` (unde `e` este obiectul exceptie).

Prinderea exceptiilor

Cand o exceptie a fost aruncata, *runtime system* incearca sa o trateze (**prinda** - *catch*). Tratarea (sau **prinderea**) unei exceptii este facuta de o portiune de cod **speciala**.

- Cum definim o astfel de portiune de cod **speciala**?
- Cum specificam faptul ca o portiune de cod speciala trateaza o **anumita** exceptie?

Sa observam urmatorul exemplu:

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

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

```
public void catchFunction() {
    try {
        f();

    } catch (Exception e) {
        System.out.println("S-a generat o exceptie");
    }
}
```

Se observa ca, daca o functie arunca o exceptie si **nu** o prinde, trebuie, in general, sa adauge **clauza `throws`** in antet.

Functia `f` va arunca intotdeauna o exceptie (din cauza ca `l` este mereu `null`). Observati cu atentie functia `catchFunction`:

- in interiorul sau a fost definit un bloc `try`, in interiorul caruia se apeleaza `f`. In general, pentru a **prinde** o exceptie, trebuie sa specificam o zona in care asteptam ca exceptia sa se produca (*guarded region*). Aceasta zona este este introdusa prin `try`.
- in continuare, avem blocul `catch (Exception e)`. La producerea exceptiei, blocul `catch` corespunzator va fi executat. In cazul nostru, se va afisa mesajul `"S-a generat o exceptie"`.

Observati urmatorul exemplu:

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

    if (l == null)
        throw new MyException();

    throw new WeirdException("This exception never gets thrown");
}

public void catchFunction() {
    try {
        f();

    } catch (MyException e) {
        System.out.println("S-a generat o exceptie Null Pointer");
    } catch (WeirdException e) {
        System.out.println("S-a generat o exceptie ciudata");
    }
}
```

In acest exemplu, functia `f` a fost modificata astfel incat sa arunce `MyException`. Observati faptul ca, in `catchFunction` avem doua blocuri `catch`. Cum exceptia aruncata de `f` este de tip `MyException`, numai primul bloc `catch` se va executa.

Prin urmare:

- putem specifica **portiuni** de cod pentru **tratarea** exceptiilor folosind blocurile `try` si `catch`;
- putem defini **mai multe** blocuri `catch` pentru a implementa o tratare **preferentiala** a exceptiilor, in functie de tipul acestora.

Nivelul la care o exceptie este tratata depinde de logica aplicatiei. Acesta **nu** trebuie sa fie neaparat nivelul imediat urmator, ce invoca sectiunea generatoare de exceptii. Desigur, propagarea de-a lungul mai multor nivele (metode) presupune utilizarea clauzei `throws`. Daca o exceptie nu este tratata nici in `main`, aceasta va conduce la **incheierea** executiei programului!

Blocuri `try-catch` imbricate

In general, vom dispune in acelasi bloc `try-catch` instructiunile care pot fi privite ca infaptuind un acelasi scop. Astfel, daca **o** operatie din secventa esueaza, se renunta la instructiunile ramase si se sare la un bloc `catch`.

Putem specifica operatii optionale, al caror esec sa **nu influenteze** intreaga secventa. Pentru aceasta folosim blocuri `try-catch` **imbricate**:

```
try {
    op1();

    try {
        op2();
        op3();

    } catch (Exception e) { ... }

    op4();
    op5();

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

Daca apelul `op2` esueaza, se renunta la apelul `op3`, se executa blocul `catch` interior, dupa care se continua cu apelul `op4`.

Blocul `finally`

Presupunem ca, in secventa de mai sus, care deschide si citeste un fisier, avem nevoie sa inchidem fisierul deschis, atat in cazul normal, cat si in eventualitatea aparitiei unei erori. In aceste conditii, se poate atasa un bloc `finally` dupa ultimul bloc `catch`, care se va executa in **ambele** cazuri mentionate.

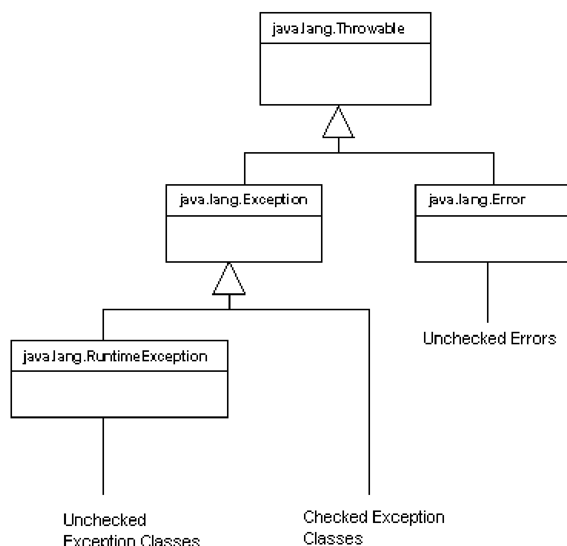
Secventa de cod urmatoare contine o structura `try`-`catch`-`finally`:

```
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 dovedeste foarte util cand in blocurile `try-catch` se gasesc insctructiuni **`return`**. El se va executa si in **acest** caz, exact inainte de executia instructiunii **`return`**, aceasta fiind executata ulterior.

Tipuri de exceptii

Nu toate exceptiile trebuie prinse cu `try-catch`. Pentru a intelege de ce, sa analizam urmatoarea clasificare a exceptiilor:



- ***checked* exceptions**, ce corespund clasei [**`Exception`**]: Acestea sunt exceptii pe care o aplicatie bine scrisa ar trebui sa le **prinda**, si sa permita **continuarea** rularii programului. Sa luam ca exemplu un program care cere utilizatorului un nume de fisier (pentru a-l deschide). In mod normal, utilizatorul va introduce un nume de fisier care exista si care poate fi deschis. Exista insa posibilitatea ca utilizatorul sa greseasca, caz in care se va arunca o exceptie `FileNotFoundException`. Un program bine scris va prinde aceasta exceptie, va afisa utilizatorului un mesaj de eroare, si ii va permite acestuia (eventual) sa reintroduca un nou nume de fisier.
- **erori**, ce corespund clasei [**`Error`**]: Acestea definesc situatii exceptionale declansate de factori **externi** aplicatiei, pe care aceasta nu le poate anticipa si nu-si poate reveni, daca se produc. Spre exemplu, tentativa de a citi un fisier care nu poate fi deschis din cauza unei defectiuni hardware (sau eroare OS), va arunca `IOError`. Aplicatia poate incerca sa prinda aceasta exceptie, pentru a anunta utilizatorul despre problema aparuta; dupa aceasta insa, programul va esua (afisand eventual *stack trace*).
- **exceptii *runtime***, ce corespund clasei [**`RuntimeException`**]: Ca si erorile, acestea sunt conditii exceptionale, insa spre **deosebire** de **erori**, ele sunt declansate de factori **interni** aplicatiei. Aplicatia nu poate anticipa, si nu-si poate reveni daca acestea sunt aruncate. **Runtime exceptions** sunt produse de diverse bug-uri de programare (erori de logica in aplicatie, folosire necorespunzatoare a unui API, etc). Spre exemplu, a realiza apeluri de metode sau membri pe un obiect `null` va produce `NullPointerException`. Fireste, putem prinde exceptia. Mai **natural** insa ar fi sa **eliminam** din program un astfel de bug care produce exceptia.

Exceptiile **checked** sunt cele **prinse** de blocurile `try-catch`. Toate exceptiile sunt **checked** cu exceptia celor de tip **`Error`**, **`RuntimeException`** si subclasele acestora.

Exceptiile **error** nu trebuie (in mod obligatoriu) prinse folosind `try-catch`. Optional, programatorul poate alege sa le prinda.

Exceptiile **runtime** nu trebuie (in mod obligatoriu) prinse folosind `try-catch`. Ele sunt de tip **`RuntimeException`**. Ati intalnit deja exemple de exceptii runtime, in urma diferitelor neatentii de programare: `NullPointerException`, `ArrayIndexOutOfBoundsException` etc. Putem arunca o `RuntimeException` fara sa o mentionam in clauza `throws` din antet:

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

Definirea de exceptii noi

Cand aveti o situatie in care alegerea unei exceptii (de aruncat) nu este evidenta, puteti opta pentru a scrie propria voastra exceptie, care sa extinda `Exception`, `RuntimeException` sau `Error`.

Exemplu:

```
class TemperatureException extends Exception {}

class TooColdException extends TemperatureException {}

class TooHotException extends TemperatureException {}
```

In aceste conditii, trebuie acordata atentie **ordinii** in care se vor defini blocurile `catch`. Acestea trebuie precizate de la clasa exceptie cea mai **particulara**, pana la cea mai **generala** (in sensul mostenirii). De exemplu, pentru a intrebuinta exceptiile de mai sus, blocul `try-catch` ar trebui sa arate ca mai jos:

```
try {
    ...

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

Afirmatia de mai sus este motivata de faptul ca intotdeauna se alege **primul** bloc `catch` care se potriveste. Un bloc `catch` referitor la o clasa exceptie **parinte**, ca `TemperatureException` prinde si exceptii de tipul claselor **copil**, ca `TooColdException`. Pozitionarea unui bloc mai general **inaintea** unuia mai particular ar conduce la **ignorarea** blocului particular.

Exceptiile in contextul mostenirii

Metodele suprascrise (overriden) pot arunca **numai** exceptiile specificate de metoda din **clasa de baza** sau exceptii **derivate** din acestea.

Exercitii

1. (**4p**) Modificati rezolvarea exercitiului 1 din laboratorul 6, astfel incat fiecare semnalare de eroare sa fie inlocuita cu aruncarea unei exceptii.
    - Alegeti exceptiile potrivite, pentru fiecare caz in parte. Care este alegerea fireasca: exceptii ***checked*** sau ***unchecked***? De ce?

      Hint: `Iterator.next`
    - Testati
2. (**2p**) Scrieti o metoda (scurta) care sa genereze `OutOfMemoryError` si o alta care sa genereze `StackOverflowError`. Verificati posibilitatea de a continua rularea dupa interceptarea acestei erori. Comparati raspunsul cu posibilitatea de a realiza acelasi lucru intr-un limbaj compilat, ce ruleaza direct pe platforma gazda (ca C).
3. (**4p**) Definiti o clasa care sa implementeze operatii pe numere intregi. Operatiile vor arunca exceptii.
    - Scrieti clasa `Calculator`, ce contine doua metode:
        - `add`: primeste doi intregi si intoarce un intreg
        - `divide`: primeste doi intregi si intoarce un intreg

        - `average`: primeste o colectie ce contine obiecte `Integer`, si intoarce media acestora. Pentru calculul mediei, sunt folosite operatiile `add` si `divide`.
    - Definiti urmatoarele exceptii si imbogatiti corespunzator definitia metodei `add`:
        - `OverflowException`: este aruncata daca suma celor doua numere depaseste `Integer.MAX_VALUE`
        - `UnderflowException`: este aruncata daca suma celor doua numere este mai mica decat `Integer.MIN_VALUE`
    - Care este alegerea fireasca: exceptii ***checked*** sau ***unchecked***? De ce? Considerati ca, pentru un utilizator care doreste efectuarea de operatii aritmetice, **singurul** mecanism disponibil este cel oferit de clasa `Calculator`.
    - Evidentiati prin teste toate cazurile posibile care genereaza exceptii.
4. (**2p**) Demonstrati intr-un program executia blocului `finally` chiar si in cazul unui `return` din metoda.

- **Solutii**

Referinte

- Exceptii pe Java Tutorials

Adus de la "http://cursuri.cs.pub.ro/~poo/wiki/index.php/Exceptii"

Vizualizări

- Pagină
- Discuţie
- Vezi sursa
- Istoric

Unelte personale

- Autentificare

Navigare

- Pagina principală
- Portalul comunităţii
- Discută la cafenea
- Schimbări recente
- Pagină aleatorie
- Ajutor

Caută

Du-te Caută

Trusa de unelte

- Ce se leagă aici
- Modificări corelate
- Trimite fişier
- Pagini speciale
- Versiune de tipărit
- Legătură permanentă
- Print as PDF

- Ultima modificare 08:19, 24 noiembrie 2012.
- Această pagină a fost vizitată de 5.117 ori.
- Conţinutul este disponibil sub GNU Free Documentation License 1.2.
- Politica de confidenţialitate
- Despre POO
- Termeni