JUnit & Excepții

- Responsabil: Andrei Casu-Pop
- Data publicării: 15.10.2017
- Data ultimei modificări: 15.10.2017

Obiective

- familiarizarea cu noţiunea de **Unit Testing** şi folosirea framework-ului **JUnit** pentru realizarea testării
- înţelegerea conceptului de **excepţie** şi utilizarea corectă a mecanismelor de **generare şi tratare** a excepţiilor puse la dispoziţie de limbajul / maşina virtuală Java

Unit testing

Unit testing-ul s-a impus în ultima perioadă în dezvoltarea proiectelor scrise în limbajul Java şi nu numai, pe măsura apariţiei unor utilitare gratuite de testare a claselor, care au contribuit la creşterea vitezei de programare şi la micşorarea semnificativă a numărului de bug-uri.

Printre avantajele folosirii framework-ului JUnit se numără:

- îmbunătăţirea vitezei de scriere a codului şi creşterea calităţii acestuia
- clasele de test sunt uşor de scris şi modificat pe măsură ce codul sursă se măreşte, putând fi compilate împreună cu codul sursă al proiectului
- clasele de test JUnit pot fi rulate automat (în suită), rezultatele fiind vizibile imediat
- clasele de test măresc încrederea programatorului în codul sursă scris şi îi permit să urmărească mai uşor cerinţele de implementare ale proiectului
- testele pot fi scrise înaintea implementării, fapt ce garantează înţelegerea funcţionalităţii de către dezvoltator

JUnit

Reprezintă un framework de Unit Testing pentru Java.

Exemplu:

Student.java

```
public class Student {

    private String name;
    private String age;

    public Student(String name, String age) {
        this.name = name;
```

```
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAge() {
        return age;
    }

    public void setAge(String age) {
        this.age = age;
    }

}
```

Group.java

```
import java.util.ArrayList;
import java.util.List;

public class Group {

    List<Student> students;

    Group () {
        students = new ArrayList<Student>();
    }

    public List<Student> getStudents() {
        return students;
    }

    public void setStudents(List<Student> students) {
        this.students = students;
    }

    public void addStudent(Student student) {
        students.add(student);
    }

    public Student getStudent(String name) {
        for (Student st : students) {
            if (null != st.getName() && st.getName().equals(name)) {
                return st;
```

```
            }
        }
        return null;
    }

    public boolean areStudentsInGroup() {
        if ( == students.size()) {
            return false;
        }
        return true;
    }

}
```

Test.java

```
import org.junit.Assert;
import org.junit.Before;

public class Test {
    private Group group;

    @Before
    public void setup() {
        group = new Group();
    }

    @org.junit.Test
    public void testNoStudentInGroup() {
        Assert.assertEquals(false, group.areStudentsInGroup());
    }

    @org.junit.Test
    public void testAddStudent() {
        Student st = new Student("Elena", "11");
        group.addStudent(st);

        Assert.assertTrue(group.getStudent("Elena").equals(st));
    }
    @After void tearDown() {
        group = null;
    }

}
```

Observaţii:

- fiecare metodă de test are adnotarea: `@Test`
- metodele de **setUp** (având adnotarea: `@Before`) şi **tearDown** (având adnotarea: `@After`) se

apelează înainte, respectiv după fiecare test. Se întrebuinţează pentru iniţializarea/eliberarea resurselor ce constituie mediul de testare, evitându-se totodată duplicarea codului şi respectându-se principiul de independenţă a testelor. Pentru exemplul dat ordinea este:
    - `@Before setUp`
    - `@Test testNoStudentInGroup`
    - `@After tearDown`
    - `@Before setUp`
    - `@Test testAddStudent`
    - `@After tearDown`
- În contextul moștenirii dacă ChildTest extends ParentTest ordinea de execuţie este urmatoarea:
    - **`ParentTest`** `@Before setUp`
    - `ChildTest @Before setUpSub`
    - `ChildTest @Test testChild1`
    - `ChildTest @After tearDownSub`
    - **`ParentTest`** `@After tearDown`
- Pentru compararea rezultatului aşteptat cu rezultatul curent se folosesc apeluri `assert`:
    - `assertTrue`
    - `assertFalse`
    - `assertEquals`
    - `assertNull` / `assertNotNull`
- **Rulare teste**: Click dreapta clasă test → *Run as* → *Junit test*
- **Rulare test**: Selectare test → click dreapta → *Run as* → *Junit test*

În cadrul laboratorului vom folosi framework-ul JUnit adăugând fişierele junit.jar şi hamcrest-all-1.3.jar cu acesta la proiectele create.

Pentru a importa un *jar* într-un proiect din `Eclipse` parcurgeţi următorii paşi: *click dreapta proiect* → *Build path* → *Configure build path* → *Libraries* → *Add jars (Add external jars)*.

Câteodată avem nevoie să testăm funcționalitatea unor clase ce folosesc metode din alte clase netestate sau care nu pot fi testate. De asemenea, există cazuri în care vrem sa testăm comportamentul clasei în situații extreme sau foarte greu de replicat (erori de disc, epuizarea spațiului pe disc, obținerea unei anumite valorii în cazul in care folosim generatoare random).

Pentru a rezolva ușor aceste necesități putem folosi obiecte de tip **mock**. Aceste obiecte simulează comporatamentul clasei mock-uite și sunt controlate din interiorul unit testelor. Pentru mai multe detalii puteți consulta pagina Mock Object.

În Java pentru a implementa mock-uri puteți folosi framework-ul **JMock**. Un scurt tutorial de instalare si folosire găsiți aici. Pentru exemple mai detaliate puteţi consulta cookbook-ul pus la dispozitie pe site-ul jmock.org.

Excepţii

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 sau din cauza unei zone invalide de pe hard-disk
- 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 == INUFFICIENT_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 (INUFFICIENT_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 (obţinut prin invocarea succesivă a metodelor din alte 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 "`S-a generat o excepţie`".

Observaţi un alt exemplu:

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

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

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

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

În acest exemplu funcţia `f` a fost modificată astfel încât să arunce `MyException`. Observaţi faptul că în `catchFunction` avem două blocuri `catch`. Cum excepţia aruncată de `f` este de tip `MyException`, numai primul bloc `catch` se va executa.

Prin urmare:

- putem specifica **porţiuni** de cod pentru **tratarea** excepţiilor folosind blocurile `try` şi `catch`
- putem defini **mai multe** blocuri catch pentru a implementa o tratare **preferenţială** a excepţiilor, în funcţie de tipul acestora

Nivelul la care o excepţie este tratată depinde de logica aplicaţiei. Acesta **nu** trebuie să fie neaparat 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 secvenţa esuează, 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.

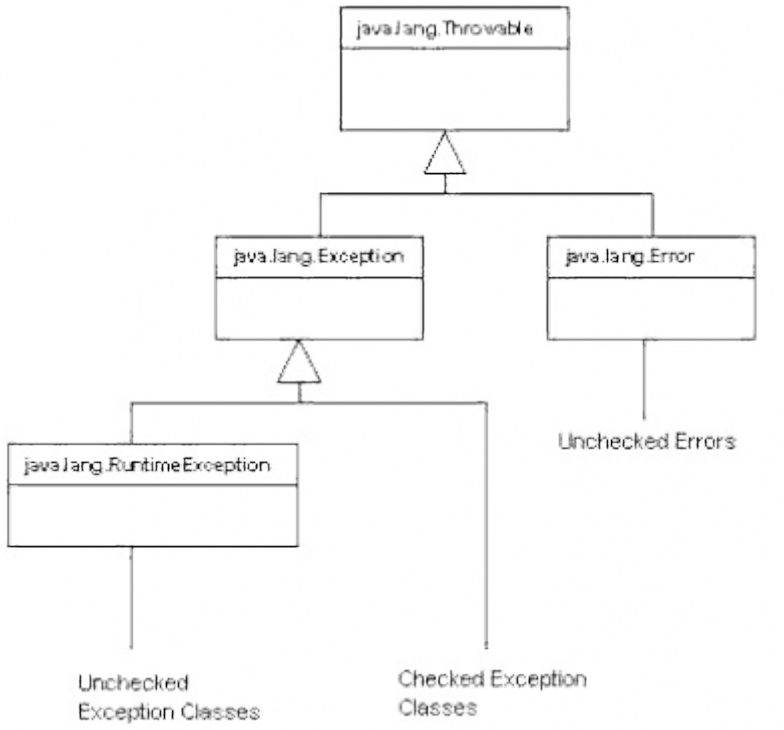
Secvenţa de cod următoare conţine o structură `try-catch-finally`:

```
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:



Checked exceptions, ce corespund clasei Exception:

- Acestea sunt excepţii pe care 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 acestuia (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, tentativa de a citi un fişier care nu poate fi deschis din cauza unei defecţiuni hardware (sau eroare OS), va arunca `IOError`.
- Aplicaţia poate încerca să prindă această excepţie, pentru a anunţa utilizatorul despre problema apărută; după această 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** cu excepţia celor de tip **Error**, **RuntimeException** şi subclasele acestora, adica cele de tip **unchecked**.

Excepţiile **error** nu trebuie (în mod obligatoriu) prinse folosind `try-catch`. Opţional, programatorul poate alege să le prindă.

Excepţiile **runtime** nu trebuie (în mod obligatoriu) prinse folosind `try-catch`. Ele sunt de tip **RuntimeException**. Aţi întâlnit deja exemple de excepţii runtime, în urma diferitelor neatenţii de programare: `NullPointerException`, `ArrayIndexOutOfBoundsException`, etc.

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 ex) {
  ...
}
```

Excepţiile în contextul moştenirii

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

Exerciţii

1. **(2p)** Creaţi clasa `Animal` şi clasa `Zoo`. Clasa `Zoo` conţine un vector de animale. Implementaţi metodele: `addAnimal(Animal a)`, `removeAnimal(Animal a)`, `boolean areAnimals()`, `getAnimals()`, `size()`. Creaţi o clasa `Test` unde veţi verifica diverse scenarii:
   - La rularea fiecărei metode veti instanţia clasa `Zoo`.
   - Metoda `testAddAnimal` - adaugă un obiect `Animal` şi verifică daca adăugarea a avut loc cu succes. Folosiţi: **`assertEquals`**
   - Metoda `testRemoveAnimal` - folosiţi **`assertTrue`**
   - Metoda `testAreAnimalsInZoo` - testul pică dacă metoda returnează `false`. *Hint*: **`Assert.fail()`**
   - Metoda `testGetAnimals` - adăugaţi două obiecte `Animal`. Verificaţi ca adăugarea a avut loc cu succes. Folosiţi **`assertFalse`**.
2. **(2p)** Acest exerciţiu urmăreşte identificarea unor cazuri de test, strict pe baza specificaţiei, în absenţa accesului la codul sursă şi a cunoaşterii modului intern de funcţionare a sistemului. Se consideră o clasa `GeometricForms` avand un constructor ce primeste un String ce poate fi unul din valorile enum-ului `Forms`.
   - Adăugaţi în build path-ul proiectului clasele din scheletul de laborator.
   - Metodele `isTriangle`, `isCircle` si `isRectangle` au drept scop evaluarea stării obiectului `GeometricForms`.
   - Creaţi un scenariu de testare pentru această clasă, prin implementarea propriilor cazuri de testare, într-o clasă `GeometricFormsTest`.
   - Construiţi teste specializate, orientate pe o anumită funcţionalitate. De exemplu, în cadrul unui test, verificaţi doar una din cele 3 metode.

   ```
   public enum Forms {
      TRIANGLE, CIRCLE, RECTANGLE
   }
   ```
3. **(2p)** Scrieţi o metodă (scurtă) care să genereze OutOfMemoryError şi o alta care să genereze StackOverflowError. Verificaţi posibilitatea de a continua rularea după interceptarea acestei

erori. Comparaţi răspunsul cu posibilitatea de a realiza acelaşi lucru într-un limbaj compilat, ce rulează direct pe platforma gazdă (ca C).

4. **(2p)** Definiţi o clasă care să implementeze operaţii pe numere **întregi**. Operaţiile vor arunca excepţii. Scrieţi clasa `Calculator`, ce conţine trei metode:
   - `add`: primeşte doi întregi şi întoarce un **întreg**
   - `divide`: primeşte doi întregi şi întoarce un **întreg**
   - `average`: primeşte o colecţie ce conţine obiecte `Integer`, şi întoarce media acestora ca un numar de tip **întreg**. Pentru calculul mediei, sunt folosite operaţiile `add` şi `divide`.
   - Definiţi următoarele excepţii şi îmbogăţiţi corespunzător definiţia metodei `add`:
   - `OverflowException`: este aruncată dacă suma celor doua numere depăşeşte `Integer.MAX_VALUE`
   - `UnderflowException`: este aruncată dacă suma celor doua numere este mai mică decat `Integer.MIN_VALUE`
   - Care este alegerea firească: excepţii **checked** sau **unchecked**? De ce? Consideraţi că, pentru un utilizator care doreşte efectuarea de operaţii aritmetice, **singurul** mecanism disponibil este cel oferit de clasa `Calculator`.
   - Evidenţiaţi prin teste toate cazurile posibile care generează excepţii.
5. **(2p)** Demonstraţi într-un program execuţia blocului `finally` chiar şi în cazul unui `return` din metoda.

Resurse

- PDF laborator
- Junit Download
- Hamcrest Core
- Schelet

Referinţe

- Mock Object
- JMock for beginners
- JMock Cookbook
- Exception
- Error
- RuntimeException
- NullPointerException
- IndexOutOfBoundsException
- NoSuchElementException
- OutOfMemoryError
- StackOverflowError