Tratarea excepţiilor

1. Scopul lucrării

Obiectivele de învățare ale acestei sesiuni de laborator sunt:
- Înțelegerea noțiunii de excepție și învățarea modului corect de folosire a excepțiilor
- Acumularea experienței de programare a excepțiilor existente în bibliotecile de clase și acelor definite de către programator

Tratarea excepţiilor constituie un mecanism care permite programelor Java să gestioneze diferitele situaţii excepţionale, cum sunt violările semantice ale limbajului şi erorile definite în programe într-un mod robust. La apariţia unei situaţii excepţionale se aruncă o excepţie. Dacă maşina virtuală Java sau mediul de execuţie detectează o violare semantică, maşina virtuală sau mediul de execuţie vor arunca implicit o excepţie. După aruncarea excepţiei controlul va fi transferat din punctul curent la o clauză **catch** corespunzătoare a blocului **try** în care a apărut excepţia. Clauza **catch** se numeşte tratarea excepţiei deoarece tratează excepţia prin executarea de acţiuni care sunt necesare recuperării din situaţia respectivă.

2. Tratarea excepţiilor

O instrucţiune **try** conţine un bloc de cod de executat. Plasarea unui bloc într-o instrucţiune **try** indică faptul că orice excepţii sau alte ieşiri anormale din bloc urmează să fie tratate corespunzător. O instrucţiune **try** poate avea orice număr de clauze **catch** opţionale care slujesc de blocuri de tratare a excepţiilor pentru blocul **try** respectiv. O instrucţiune **try** poate avea şi o clauză **finally**. Blocul **finally** este executat întotdeauna înainte de părăsirea instrucţiunii **try**; el "face curăţenie" după blocul **try**. Remarcaţi că o instrucţiune **try** trebuie fie să aibă o clauză **catch** fie o clauză **finally**, fie ambele.
Iată un exemplu de instrucţiune **try** care are o clauză **catch** şi una **finally**:

```
try
{
    out.write(b);
}
catch (IOException e)
{
    System.out.println("Output Error");
}
finally
{
    out.close();
}
```

Dacă `out.write()` aruncă o `IOException`, excepţia este interceptată de clauza **catch**. Indiferent dacă `out.write()` revine normal sau aruncă o excepţie, blocul **finally** este executat, ceea ce asigură că `out.close()` este apelată întotdeauna.
Instrucţiunea **try** execută blocul care urmează după cuvântul cheie **try**. Dacă este aruncată o excepţie din blocul **try** şi instrucţiunea **try** are clauze **catch**, atunci respectivele clauze vor fi cercetate, în ordine, pentru a găsi una care poate trata excepţia. Dacă există o clauză **catch** care poate trata excepţia, atunci se execută respectivul bloc **catch**.
Dar dacă instrucţiunea **try** nu are nici o clauză **catch** care poate trata excepţia (sau nu are de loc clauze **catch**), excepţia se propagă mai departe în blocurile exterioare celui în care a apărut în metoda curentă. Dacă metoda curentă nu conţine o instrucţiune **try** cu un bloc de tratare corespunzător, excepţia se propagă în sus spre metoda apelantă. Dacă această metodă nu conţine o instrucţiune **try** cu un bloc de tratare corespunzător, excepţia se propagă iarăşi în sus. În sfârşit, dacă

nu se găseşte o instrucţiune **try** cu bloc de tratare corespunzător, programul în curs de execuţie se termină.
Clauza **catch** se declară cu un parametru care specifică ce tip de excepţie poate trata. Parametrul dintr-o clauză **catch** trebuie să fie de tipul **Throwable** sau al uneia dintre subclasele sale. La apariţia unei excepţii, clauzele **catch** sunt cercetate pe rând pentru a o găsi pe prima care are parametrul de acelaşi fel ca tipul excepţiei aruncate sau este o superclasă a excepţiei aruncate. La executarea blocului **catch** corespunzător, obiectul excepţie actual este transmis ca argument al blocului **catch**. Codul din blocul **catch** trebuie să facă ceea ce este necesar pentru a trata situaţia excepţională.

Clauza **finally** a unei instrucţiuni **try** este executată întotdeauna, indiferent de modul în care s-a părăsit instrucţiunea **try**. De aceea clauza respectivă este un loc potrivit pentru a efectua operaţii de "curăţare", cum sunt*: închiderea fişierelor, eliberarea resurselor şi închiderea conexiunilor de reţea*.

3. Declararea excepţiilor

Dacă se aşteaptă ca o metodă să arunce excepţii, atunci metoda respectivă trebuie să declare acest lucru în clauza **throws**. Dacă implementarea unei metode conţine o instrucţiune **throw**, atunci este posibil să fie aruncată o excepţie din metodă. În plus, dacă metoda apelează o alta declarată cu clauză **throws**, există posibilitatea să fie aruncată o excepţie din metoda respectivă. Dacă excepţia nu este interceptată în metoda respectivă într-un **try-catch,** atunci ea va fi aruncata afară spre apelant. Orice excepţie care poate fi aruncată în afara unei metode în acest mod, trebuie să fie listată într-o clauză **throws** în declaraţia metodei. Clasele listate într-o clauză **throws** trebuie să fie **Throwable** sau subclasa a sa; clasa **Throwable** este superclasa tuturor obiectelor care pot fi aruncate în Java.

Există anumite tipuri de **Throwable** care nu trebuie listate în clauze **throws**. Mai specific, dacă excepţia este o instanţă a lui **Error**, **RunTimeException**, sau o subclasă a acestora, atunci nu trebuie listată în clauza **throws**. Subclasele clasei **Error** corespund unor situaţii care nu pot fi prevăzute uşor, cum este epuizarea memoriei disponibile. Subclasei lui **RunTimeException** corespund mai multor probleme uzuale care apar la execuţie, cum sunt conversiile/forţările (casts) de tip şi problemele legate de indecşii tablourilor. Motivul tratării speciale a acestora este că ele pot fi aruncate dintr-un număr atât de mare de locuri încât aproape fiecare metodă ar trebui să le declare.
Consideraţi următorul exemplu:

```
import java.io.IOException;
class throwsExample
{
    char[] a;
    int position;
    ...
    // Method explicitly throws an exception
    int read() throws IOException
   {
        if (position >= a.length)
            throw new IOException();
        return a[position++];
    }
    // Method implicitly throws an exception
    String readUpTo(char terminator) throws IOException
    {
        StringBuffer s = new StringBuffer();
        while (true)
       {
            int c = read(); // Can throw IOException
            if (c == -1 || c == terminator)
           {
                return s.toString();
           }
            s.append((char)c);
        }
```

```
        return s.toString();
    }
    // Method catches an exception internally
    int getLength()
   {
        String s;
        try
        {
            s = readUpTo(':');
        }
        catch (IOException e)
        {
            return 0;
        }
        return s.length();
    }
    // Method can throw a RunTimeException
    int getAvgLength()
    {
        int count = 0;
        int total = 0;
        int len;
        while (true)
        {
            len = getLength();
            if (len == 0)
                break;
            count++;
            total += len;
        }
        return total/count; // Can throw ArithmeticException
    }
}
```

Metoda `read()` poate arunca o **IOException**, aşa că declară acest lucru în clauza sa **throws**. Fără o clauză **throws**, compilatorul ar avertiza că metoda trebuie fie să declare **IOException** în clauza sa **throws** fie să o intercepteze. Deşi metoda `readUpTo()` nu aruncă explicit nici o excepţie, ea apelează metoda `read()` care aruncă o **IOException**, aşa că declară aceasta în clauza sa **throws**. Fie excepţia aruncată implicit sau explicit, cerinţa ca să fie declarată sau interceptată există. Metoda `getLength()` interceptează **IOException** aruncată de `readUpTo()`, aşa că nu trebuie să o declare. Ultima metodă, `getAvgLength()`, poate arunca o **ArithmeticException** dacă variabila `count` are valoarea zero. Deoarece **ArithmeticException** este o subclasă a lui **RuntimeException**, faptul că ea poate fi aruncată afară din `getAvgLength()` nu trebuie declarat în clauza **throws**.

4. Generarea (aruncarea) excepţiilor

Un program Java poate folosi mecanismul de tratare a excepţiilor pentru a trata erori de program anume de o manieră curată. Pur şi simplu programul foloseşte instrucţionea **throw** pentru a semnala excepţia. Instrucţiunea **throw** trebuie urmată de un obiect de tipul **Throwable** sau al unei subclase a sa. Pentru excepţiile definite prin program, dorim de obicei ca obiectul excepţie să fie o instanţa a unei subclase a clasei **Exception**. În majoritatea cazurilor are sens să definim o nouă subclasă a lui **Exception,** specifică programului nostru.
Consideraţi exemplul care urmează:

```
class WrongDayException extends Exception
{
    public WrongDayException () {}
    public WrongDayException(String msg)
   {
        super(msg);
    }
}
```

```
public class ThrowExample
{
    void doIt() throws WrongDayException
   {
        int dayOfWeek =(new java.util.Date()).getDay();
        if (dayOfWeek != 2  && dayOfWeek != 4)
            throw new WrongDayException("Tue. or Thur.");
        // The rest of doIt's logic goes here
        System.out.println("Did it");
    }
    public static void main (String [] argv)
   {
        try
       {
            (new ThrowExample()).doIt();
        }
        catch (WrongDayException e)
       {
            System.out.println("Sorry, can do it only on "
                               + e.getMessage());
        }
    }
}
```

Codul din acest exemplu defineşte o clasă numită `WrongDayException` pentru a reprezenta tipul de excepţie specific aruncat în exemplu. Clasa **Throwable** şi majoritatea subclaselor sale au cel puţin doi constructori. Un constructor ia ca argument un **String** folosit ca mesaj textual care explică excepţia, iar celălalt nu ia nici un argument. De aceea clasa `WrongDayException` defineşte doi constructori.
În clasa `ThrowExample`, dacă ziua curentă nu este nici Tuesday (marţi) nici Thursday (joi), metoda `doIt()` aruncă o `WrongDayException`. Remarcaţi că obiectul `WrongDayException` este creat la momentul aruncării sale. Este o practică des întâlnită să se ofere informaţie despre excepţie la aruncarea sa, aşa că se foloseşte un argument şir de caractere în instrucţiunea de alocare a lui `WrongDayException`. Declaraţia metodei `doIt()` conţine o clauză **throws** pentru a indica faptul că ea poate arunca o `WrongDayException`.
Metoda `main()` din `ThrowExample` include apelul la metoda `doIt()` într- instrucţiune **try**, astfel că ea poate intercepta orice `WrongDayException` aruncat de către `doIt()`. Blocul **catch** afişează un mesaj de eroare folosind metoda `getMessage()` din obiectul excepţie. Această metodă regăseşte şirul transmis constructorului la crearea obiectului excepţie.

1.1. Afişarea trasărilor de stivă

La interceptarea unei excepţii poate fi util să tipărim o trasare a stivei pentru a ne da seama de unde provine excepţia. O trasare a stivei arată în modul următor:

```
java.lang.ArithmeticException: / by zero
      at t.cap(t.java:16)
      at t.doit(t.java:8)
      at t.main(t.java:3)
```

Putem tipări trasarea stivei apelând metoda **printStackTrace()** pe care toate obiectele **Throwable** o moştenesc din clasa **Throwable**. Spre exemplu:

```
int cap (x) {return 100/x}
try
{
    cap(0);
}
catch(ArithmeticException e)
{
    e.printStackTrace();
```

```
}
```

Putem tipări trasarea stivei oriunde în aplicaţii fără a arunca de fapt o excepţie. Exemplu:

```
new Throwable().printStackTrace();
```

1.2. Re-aruncarea excepţiilor

După interceptarea unei excepţii ea poate fi re-aruncată dacă este cazul. Decizia pe care trebuie să o luăm la re-aruncarea unei excepţii este despre locaţia de unde trasarea stivei să spună ca a fost aruncat obiectul. Putem face ca excepţia re-aruncată să pară a fi fost aruncată din locaţia excepţiei iniţiale sau din locaţia re-aruncării curente.
Pentru locaţia iniţială, tot ce trebuie făcut este să re-aruncăm excepţia:

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

Pentru a arăta locaţia reală e nevoie să apelăm metoda **fillInStackTrace()** a excepţiei. Această metodă setează informaţia din excepţie pe baza contextului de execuţie curent. Iată un exemplu care foloseşte metoda **fillInStackTrace()**:

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

Este important să apelăm **fillInStackTrace()** pe aceeaşi linie cu instrucţiunea **throw**, astfel ca numărul precizat de trasarea stivei să corespundă cu linia în care apare instrucţiunea **throw**. Metoda **fillInStackTrace()** întoarce o referinţa la clasa **Throwable**, aşa că este nevoie să forţăm tipul (cast) referinţei la tipul real al excepţiei.

1.3. Potrivirea exceptiilor

La o clauza **try** ne putem define mai multe clause **catch** care sa trateze excaptii de tip diferit. Cand o exceptie este aruncata, mediul Java va incerca sa gaseasca clauza **catch** care sa se potriveasca tipului exceptiei aruncate. Daca nu gaseste nici una, va incerca pentru supertipul exceptiei. Daca nu a fost gasita nici o astfel de clauza catch care sa potriveasca supertipului exceptiei, atunci exceptia se va propaga in jos in stiva de apeluri.Acest process se numeste potrivirea exceptiilor. Un exemplu de aces fel este prezentat mai jos:

```
import java.io.*;
public class CitireDate {
public static void main(String args[]) {
    try {
       RandomAccessFile pf = new RandomAccessFile("myfile.txt", "r");
       byte v[] = new byte[500];
       pf.readFully(v, 0, 500);
    }
    catch(FileNotFoundException e) {
```

```
            System.err.println("Fisierul nu a fost gasit);
            System.err.println(e.getMessage());
            e.printStackTrace();
        }
        catch(IOException e) {
            System.err.println("Eroare de tip IO Error");
            System.err.println(e.toString());
            e.printStackTrace();
            }
        }
}
```

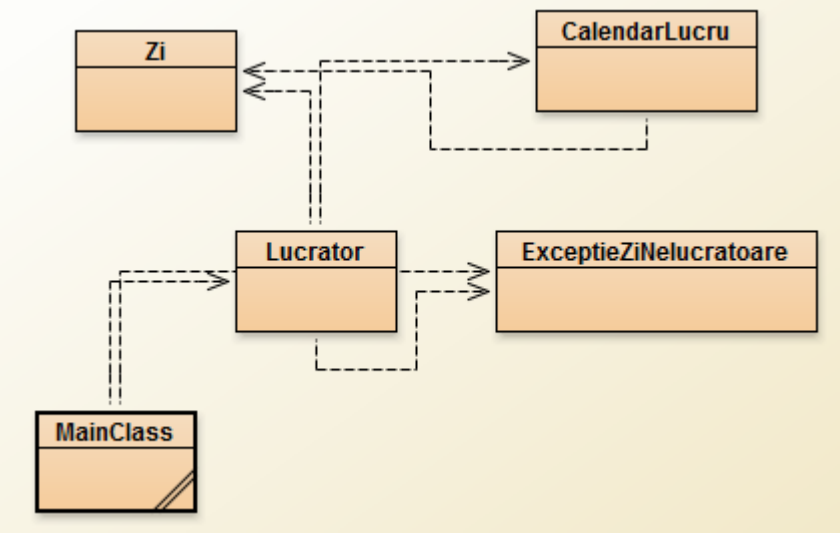
5. Sfaturi referitoare la excepţii

Folosiţi excepţiile pentru:

1. A trata problemele la nivelul corespunzător. (Evitaţi să interceptaţi excepţiile în afara cazului că ştiţi ce să faceţi cu ele).
2. Rezolvaţi problema şi apelaţi din nou metoda care a cauzat excepţia.
3. Corectaţi lucrurile şi continuaţi fără a re-încerca metoda.
4. Calculaţi un rezultat alternativ în locul celui care ar fi trebuit produs de către metodă.
5. Faceţi tot ce puteţi în contextul curent şi apoi re-aruncaţi aceeaşi excepţie spre un context mai sus.
6. Faceţi tot ce puteţi în contextul curent şi apoi aruncaţi o excepţie diferită spre un context mai sus.
7. Terminaţi execuţia programului.
8. Simplificaţi. (Dacă schema de excepţii a Dvs. complică lucrurile, atunci e dificil şi enervant de folosit.)
9. Faceţi-vă bibliotecile şi programele mai sigure. (Aceasta este o investiţie pe termen scurt d.p.d.v. al depanării şi una pe termen lung d.p.d.v. al robusteţii aplicaţiei.)

6. Mersul lucrării

6.1. Implementati o aplicație pentru a simula activitatea de lucru pentru angajati conform unui calendar pre-stabilit. Fiecare lucrator va fi identificat printr-un nume si va avea un calendar de lucru in care se vor specifica zilele lucratoare si nelucratoare. Atunci cand un lucrator este pus la lucru intr-o zi nelucratoare, se va genera o exceptie. Diagrama de clase este prezentată mai jos:



Detalii de implementare:

6.1.1. Clasa **Zi** va implementa o zi oarecare din saptamana, fiind caracterizata de un nume (Luni, Marti, Miercuri ….) si de o proprietate in care se specifica daca ziua este sau nu lucratoare. Initializarea, modificarea si accesul la cele doua proprietati se va face prin intermediul contructorilor, metodelor setter si metodelor getter (exemplu void setNume(String nume), String getNume(), void setLucratoare(), void setNelucratoare(), boolean esteLucratoare() etc…

6.1.2. Clasa **CalendarLucru** va contine un vector de zile. Implicit aceasta clasa trebuie sa specifice toate zilele saptamanii, iar zilele de sambata si duminica sa fie setate ca zile "ne lucratoare". Modificarea valorilor se va face, la fel, prin intermediul metodelor setter si getter.

6.1.3. Clasa **Lucrator**, va descrie un lucrator oarecare. Fiecare lucrator va fi indentificat printr-un nume (numele lucratorului) si un calendar de lucru – de tip **CalendarLucru**. Clasa Lucrator va implementa metoda **public void lucreaza(String zi)**, care va printa un mesaj daca ziua data ca argument este una lucratoare si va arunga o exceptie in cazul cand ziua corespunzatoare din calendar este "nelucratoare".

6.1.4. **ExceptieZiNelucratoare** este clasa in care se va implementa o exceptie (extinde clasa Exception), cu un mesaj de exceptie personalizat.

6.1.5. In calsa **MainClass**, metoda … main(…) se vor crea lucratori, calendare de lucru, se vor asigna la fiecare lucrator un calendar de lucru si se vor pune lucratorii la munca. Atunci cand un lucrator trebuie sa lucreze intr-o zi nelucratoare, se va genera o exceptie de tip **ExceptieZiNelucratoare**. Aceste exceptii vor trebui tratate corespunzator cu try{..} - catch(…).

6.1.6. Exemplu rulare, pentru un lucrator **Lucrator l**, cu numele „Gigel", pentru care zilele luni – vineri sunt lucratoare iar zilele sambata si duminica sunt zile nelucratoare, pentru un apel de metode:

```
l.lucreaza("luni");
l.lucreaza("marti");
l.lucreaza("Luni");
l.lucreaza("duminica");
```

se va afisa în linia de comandă:

```
Lucratorul Gigel lucreaza Luni
Lucratorul Gigel lucreaza Marti
Lucratorul Gigel lucreaza Luni
Exceptie: lucru in zi nelucratoare
```

6.2. Definiţi o referinţa la un obiect şi iniţializaţi-o la **null**. Încercaţi să apelaţi o metodă folosind această referinţă. Apoi includeţi codul într-o clauză **try-catch** pentru a intercepta excepţia.

6.3. Scrieti cod care sa genereze si sa trateze cu clauza **catch** o eroare de tip **ArrayIndexOutOfBoundsException.**

6.4. Creaţi o clasă cu o metodă **main()** care aruncă un obiect de clasa **Exception** în interiorul unui bloc **try**. Daţi constructorului pentru **Exception** un argument **String**. Interceptaţi excepţia într-o clauză **catch** şi afişaţi argumentul **String**. Adăugaţi o clauză **finally** şi tipăriţi un mesaj pentru a dovedi că s-a ajuns acolo.

6.5. Creaţi propria Dvs. clasă excepţie folosind cuvântul cheie **extends**. Scrieţi un constructor pentru clasă care ia ca argument un **String** şi îl stochează în obiect într-o referinţa la **String**. Scrieţi o metodă care afişează şirul stocat. Creaţi o clauză **try-catch** pentru a exersa noua excepţie.

6.6. Creaţi o clasă cu două metode, **f()** şi **g()**. În **g()**, aruncaţi o excepţie de un tip nou, definit de Dvs.. În **f()**, apelaţi **g()**, interceptaţi excepţia sa şi, în clauza **catch**, aruncaţi o excepţie diferită (de un al doilea tip, definit de Dvs.). Testaţi-vă codul în **main()**.

6.7. Creaţi o ierarhie de excepţii cu trei nivele. Apoi creaţi o clasă de bază **A** cu o metodă care aruncă o excepţie la baza ierarhiei definite de Dvs. Moşteniţi **B** din **A** şi suprascrieţi metoda astfel ca metoda să arunce excepţie la nivelul al doilea al ierarhiei. Repetaţi procedeul moştenind clasa **C** din **B**. În **main()**, creaţi **C** şi convertiţi-o (upcast) la **A**, apoi apelaţi metoda.