TEHNICI AVANSATE DE PROGRAMARE
LUCRARE DE LABORATOR 5

Excepţii în Java. Aplicaţii

I. SCOPUL LUCRĂRII

Lucrarea de faţă are rolul de a prezenta şi familiariza studentul cu noţiuni privind excepţiile şi tratarea acestora în limbajului Java.

La sfârşitul acestei lucrări, studentul va avea posibilitatea să scrie programe Java în care să folosească noţiunile legate de excepţii precum şi tratarea acestora.

II. NOŢIUNI TEORETICE

Observaţie: Vezi noţiunile prezentate în cursul 3.

1. Tratarea excepţiilor

Tratarea excepţiilor se realizează prin intermediul blocurilor de instrucţiuni try, catch şi finally. O secvenţă de cod care tratează anumite excepţii trebuie să arate astfel:

```
try {
// Instructiuni care pot genera exceptii
}
catch (TipExceptie1 variabila) {
// Tratarea exceptiilor de tipul 1
}
catch (TipExceptie2 variabila) {
// Tratarea exceptiilor de tipul 2
}
. . .
finally {
// Cod care se executa indiferent
// daca apar sau nu exceptii
}
```

Să considerăm următorul exemplu: citirea unui fişier octet cu octet şi afisarea lui pe ecran. Fără a folosi tratarea excepţiilor metoda responsabilă cu citirea fişierului ar arăta astfel:

```
public static void citesteFisier(String fis) {
FileReader f = null;
// Deschidem fisierul
System.out.println("Deschidem fisierul " + fis);
f = new FileReader(fis);
// Citim si afisam fisierul caracter cu caracter
int c;
while ( (c=f.read()) != -1)
System.out.print((char)c);
// Inchidem fisierul
System.out.println("\\nInchidem fisierul " + fis);
f.close();
}
```

Observatii:

Această secvenţă de cod va furniza erori la compilare deoarece în Java tratarea erorilor este obligatorie. Folosind mecanismul excepţiilor metoda citeste îşi poate trata singură erorile care pot surveni pe parcursul execuţiei sale.

Mai jos este codul complte şi corect al unui program ce afişează pe ecran conţinutul unui fişier al cărui nume este primit ca argument de la linia de comandă. Tratarea excepţiilor este realizată complet chiar de către metoda citeste.

```
import java .io .*;
public class CitireFisier {
        public static void citesteFisier ( String fis) {
        FileReader f = null ;
        try {
                // Deschidem fisierul
                System . out. println (" Deschidem fisierul " + fis);
                f = new FileReader (fis );
                // Citim si afisam fisierul caracter cu caracter
                int c;
                while ( (c=f. read ()) != -1)
                        System . out. print (( char )c);
        } catch ( FileNotFoundException e) {
                // Tratam un tip de exceptie
```

```
                System . err. println (" Fisierul nu a fost gasit !");
                System . err. println (" Exceptie : " + e. getMessage ());
                System . exit (1);
        } catch ( IOException e) {
                // Tratam alt tip de exceptie
                System . out. println (" Eroare la citirea din fisier !");
                e. printStackTrace ();
        } finally {
                if (f != null ) {
                // Inchidem fisierul
                System . out. println ("\ nInchidem fisierul .");
        try {
                f. close ();
        } catch ( IOException e) {
                System . err. println (" Fisierul nu poate fi inchis !");
                e. printStackTrace ();
                }
        }
        }
}
public static void main ( String args []) {
        if ( args . length > 0)
                citesteFisier ( args [0]) ;
        else
                System . out. println (" Lipseste numele fisierului !");
        }
}
```

2. “Aruncarea” excepţiilor

În exemplul de mai sus dacă nu facem tratarea excepţiilor în cadrul metodei citeste atunci metoda apelantă (main) va trebui să facă acest lucru:

```
import java .io .*;
public class CitireFisier {
    public static void citesteFisier ( String fis)
    throws FileNotFoundException , IOException {
        FileReader f = null ;
        f = new FileReader (fis );
        int c;
        while ( (c=f. read ()) != -1)
            System . out. print (( char )c);
        f. close ();
        }
    public static void main ( String args []) {
        if ( args . length > 0) {
        try {
            citesteFisier ( args [0]) ;
        } catch ( FileNotFoundException e) {
            System . err. println (" Fisierul nu a fost gasit !");
            System . err. println (" Exceptie : " + e);
        } catch ( IOException e) {
            System . out. println (" Eroare la citirea din fisier !");
            e. printStackTrace ();
            }
        } else
            System . out. println (" Lipseste numele fisierului !");
    }
}
```

Observaţi că, în acest caz, nu mai putem diferenţia excepţiile provocate de citirea din fişier şi de inchiderea fişierului, ambele fiind de tipul IOException.

De asemenea, inchiderea fişierului nu va mai fi facută în situatia în care apare o excepţie la citirea din fişier. Este situaţia în care putem folosi blocul finally fără a folosi nici un bloc catch:

```
public static void citesteFisier(String fis)
```

```
throws FileNotFoundException, IOException {
        FileReader f = null;
        try {
                f = new FileReader(numeFisier);
                int c;
                while ( (c=f.read()) != -1)
                        System.out.print((char)c);
        }
        finally {
                if (f!=null)
                f.close();
        }
}
```

Metoda apelantă poate arunca la rândul sâu excepţiile mai departe către metoda care a apelat-o la rândul ei. Această înlănţuire se termină cu metoda main care, dacă va arunca excepţiile ce pot apărea în corpul ei, va determina trimiterea excepţiilor către maşina virtuală Java.

Tratarea excepţiilor de către JVM se face prin terminarea programului şi afişarea informaţiilor despre excepţia care a determinat acest lucru.

Exemplu: Comuniacarea client/server prin datagrame

```
/*
 * Server
 */
public class DatagramServer {
    public static final int PORT = 8200;
    private DatagramSocket socket = null ;
    DatagramPacket cerere , raspuns = null ;

    public void start () throws IOException {
        socket = new DatagramSocket ( PORT );
        try {
            while ( true ) {
            // Declaram pachetul in care va fi receptionata cererea
            byte [] buf = new byte [256];
            cerere = new DatagramPacket (buf , buf. length );
            System . out. println (" Asteptam un pachet ... ");
            socket . receive ( cerere );
            // Aflam adresa si portul de la care vine cererea
            InetAddress adresa = cerere . getAddress ();
            int port = cerere . getPort ();
            // Construim raspunsul
            String mesaj = " Hai " + new String ( cerere . getData ());
            buf = mesaj . getBytes ();
```

```
                // Trimitem un pachet cu raspunsul catre client
                raspuns = new DatagramPacket (buf , buf.length , adresa , port );
                socket . send ( raspuns );
                }
            }
            finally {
                if ( socket != null )
                    socket . close ();
                }
        }

        public static void main ( String [] args ) throws IOException {
            new DatagramServer (). start ();
        }
}

/*
 * Client
 */

import java . net .*;
import java .io .*;

public class DatagramClient {

        public static void main ( String [] args ) throws IOException {
            // Adresa IP si portul la care ruleaza serverul
            InetAddress adresa = InetAddress . getByName ("127.0.0.1");
            int port =8200;
            DatagramSocket socket = null ;
            DatagramPacket packet = null ;
            byte buf [];
            try {
                // Construim un socket pentru comunicare
                socket = new DatagramSocket ();
                // Construim si trimitem pachetul cu cererea catre server
                buf = " Craiova ". getBytes ();
                packet = new DatagramPacket (buf , buf.length , adresa , port );
                socket . send ( packet );
                // Asteaptam pachetul cu raspunsul de la server
                buf = new byte [256];
                packet = new DatagramPacket (buf , buf. length );
                socket . receive ( packet );
                // Afisam raspunsul (" Hai Craiova !")
                System . out. println ( new String ( packet . getData ()));
            }
            finally {
                if ( socket != null )
                    socket . close ();
                }
            }
}
```

- **Precizări:**

O datagramă, in Java, este reprezentata printr-un obiect din clasa *DatagramPacket*. Rutarea datagramelor de la o maşină la alta se face exclusiv pe baza informaţiilor conţinute de acestea. Primirea şi trimiterea datagramelor se realizează prin intermediul unui socket, modelat prin intermediul clasei *DatagramSocket*.

Clasa *DatagramPacket* conţine următorii constructori:

DatagramPacket(byte[] buf, int length, InetAddress address, int port)
DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)
DatagramPacket(byte[] buf, int offset, int length, SocketAddress address)
DatagramPacket(byte[] buf, int length, SocketAddress address)
DatagramPacket(byte[] buf, int length)
DatagramPacket(byte[] buf, int offset, int length)

Primele două perechi de constructori sunt pentru creare pachetelor ce vor fi expediate, diferenţa între ele fiind utilizarea claselor *InetAddress*, respectiv *SocketAddress* pentru specificarea adresei desinaţie. A treia pereche de constructori este folosită pentru crearea unui pachet în care vor fi recepţionate date, ei nespecificând vreo sursă sau destinaţie.

După crearea unui pachet procesul de trimitere şi primire a acestuia implică apelul metodelor send şi receive ale clasei *DatagramSocket*. Deoarece toate informaţiile sunt incluse în datagramă, acelaşi socket poate fi folosit atât pentru trimiterea de pachete, eventual către destinaţii diferite, cât şi pentru recepţionarea acestora de la diverse surse. In cazul în care refolosim pachete, putem schimba conţinutul acestora cu metoda *setData()*, precum şi adresa la care le trimitem prin *setAddress(), setPort()* ,si *setSocketAddress()*.

Extragerea informaţiilor conţiunte de un pachet se realizează prin metoda *getData()* din clasa *DatagramPacket*. De asemenea, această clasă oferă metode pentru aflarea adresei IP şi a portului procesului care a trimis datagrama, pentru a-i putea răspunde dacă este necesar. Acestea sunt: *getAdress(), getPort()* ,si *getSocketAddress()*.

3. Introducerea datelor de la tastatură

O metodă pentru introducerea datelor de la tastatură ar fi:

```
static String citireSir( ) throws IOException{
        InputStreamReader f=new InputStreamReader(System.in);
        BufferedReader g=new BufferedReader(f);
        String h;
        h=g.readLine( );
        return h;
}
```

- **Precizări:**

În corpul metodei *citireSir* apar instrucţiuni de lucru cu aşa numitele fluxuri. Un flux (stream) poate fi flux de intrare sau flux de ieşire. Un flux de intrare este o cale pe care o parcurg datele de la o sursă de date la spaţiul din memoria internă a calculatorului unde se înregistrează valorile variabilelor programului care solicită datele, iar un flux de ieşire este o cale pe care o parcurg datele de la spaţiul din memoria internă rezervat variabilelor unui program spre o destinaţie

unde se înregistrează datele.
Ca urmare a apelului metodei *citesteSir()*, se creează fluxul necesar pentru a prelua un şir de la tastatură şi apoi se trece la execuţia metodei *readLine( )*. Metoda *readLine( )* este o metodă care aparţine unei clase din Java API. În momentul în care se începe execuţia ei, programul trece în starea de aşteptare a unui şir de caractere.

Şirul de caractere pe care-l tastăm este preluat de program după ce acţionăm tasta *Enter*. Se impune să precizăm că şirul tastat de noi este preluat de program şi dacă în locul tastei Enter tastăm Alt-13.

După ce tastăm Enter, sau Alt-13, şirul introdus de noi este preluat ca valoare a variabilei h din metoda *citireSir( )* şi returnat. Ca urmare, variabila s din instrucţiunea *s=citireSir( )* va avea ca valoare şirul tastat de noi.

Valoarea variabilei s este convertită într-un număr întreg de instrucţiunea:

a=Integer.valueOf(s).intValue( );

Şirul pe care îl introducem de la astatură trebuie să fie un număr întreg. Dacă nu respectăm această condiţie, atunci Java emite un mesaj de eroare şi abandonează execuţia (vezi cursul 3: Excepţii şi tratarea acestora în Java).

Pe parcurs vom avea şi probleme în care trebuie să introducem numere reale pe care le vom asocia unor variabile de tip float. În acest caz trebuie să folosim pentru conversia de la un şir s la un număr real r instrucţiunea:

r=Float.valueOf(s).floatValue( );

Exemplu util:

```
import java.io.*;

class Putere
{
    public static void main (String[] args) throws IOException
    {
       InputStreamReader inStream =
       new InputStreamReader( System.in ) ;
       BufferedReader stdin =
       new BufferedReader( inStream );

       String inData;
       int num, square; // declaram doua variabile intregi
       System.out.println("Introduceti un intreg:");
       inData = stdin.readLine();

       num = Integer.parseInt( inData ); // se converteste inData la intreg
       square = num * num ; // calculul puterii
       System.out.println("Patratul lui " + inData + " este " + square);
    }
}
```

III. MODUL DE LUCRU

- Clasic:

1. Se editează codul sursă al programului Java folosind un editor de text disponibil (de ex., se poate utiliza Notepad).
2. Se salvează fişierul cu extensia **.java**.
3. Compilarea aplicaţiei Java se va face din linia de comandă:
   javac nume_fişier_sursă.java
   În cazul în care programul conţine erori acestea vor fi semnalate şi afişate.
4. Pentru rularea aplicaţiei Java, se lansează interpretorul Java:
   java nume_clasă_care_conţine_main

- Se foloseşte utilitarul disponibil în laborator J2SDK Net Beans IDE.

IV. TEMĂ

- Se vor parcurge toate exemplele prezentate în platforma de laborator testându-se practic (acolo unde este cazul).

- Implementaţi o clasa ce descrie parţial o stivă de numere întregi cu operaţiile de adăugare a unui element, respective de scoatere a elementului din vârful stivei. Dacă presupunem că stiva poate memora maxim 100 de elemente, ambele operaţii pot provoca excepţii. Pentru a personaliza aceste excepţii se va crea o clasă specifică denumită *ExceptieStiva*.

- Implementaţi o aplicaţie client/server pentru comunicarea cu socket-uri(socluri de comunicatie).