Lucrarea 6

Tratarea excepţiilor

Cuprins

Pe parcursul execuţiei unui program pot apărea evenimente deosebite care îi modifică parcursul normal. Asemenea evenimente pot fi situaţii de eroare (de exemplu încercarea de a accesa un element de tablou aflat dincolo de limitele tabloului) sau pur şi simplu situaţii care necesită o altă abordare decât cea obişnuită (de exemplu, la parcurgerea secvenţială a unui fişier de intrare, la un moment dat se atinge sfârşitul fişierului; acest eveniment, fără a fi o eroare, va schimba cursul de până atunci al programului). Evenimentele de genul celor amintite se mai numesc excepţii.

O excepţie este un obiect ce aparţine clasei predefinite *Throwable* sau unei clase descendente din *Throwable*. Clasa *Throwable* este definită în pachetul *java.lang*. Limbajul *Java* dispune, de fapt, de o întreagă ierarhie de clase predefinite pentru reprezentarea excepţiilor. Programatorul poate utiliza clasele predefinite sau poate să-şi definească propriile clase de excepţii, cu condiţia ca ele să descindă direct sau indirect din *Throwable*. Prin convenţie, clasele definite de utilizatori pentru reprezentarea excepţiilor vor fi derivate din clasa *Exception*.

Definirea unui tip de excepţie de către utilizator se justifică de obicei prin:

- necesitatea reţinerii de date suplimentare şi/sau mesaje mai detaliate despre excepţie;
- necesitatea ca dintr-o clasă mai largă de excepţii să fie captate doar o anumită parte..

Tratarea clasică a excepţiilor

Se va considera un exemplu în care se va face abstracţie de faptul că limbajul *Java* dispune de un mecanism special de tratare a excepţiilor:

```
class Tabel{
        public static boolean fan_err = false;
        public static int Increm(int[ ] tab,int indice,int val) {
            if(indice>=0 && indice<tab.length){
                    tab[indice]+=val;
                    fan_err = false;
                    return tab[indice];
            }
            else{
                    fan_err = true;
```

```
                    return 0;
                }
        }

        public static int indexOf(int[ ] tab, int val) {
            for(int i = 0;i<tab.length;i++)
                if(tab[i]==val) return i;
            return -1;
        }
}

class ClientTabel{
        public static void main(String[ ] arg) {
            int[ ] tab = new int[10];
            //se completeaza tabloul cu valori
            int suma=0,indice,val;
            boolean gata=false;
            do{
                //se citesc de la tastatura un indice si o valoare
                suma+=Tabel.Increm(tab,indice,val);
                if(Tabel.fan_err) //a fost eroare...
                else //merg mai departe
                    if(Tabel.indexOf(tab,val)<0) //valoarea val nu exista in tablou
                    else //valoarea val exista
                    //. . .
            }while(!gata);
        }
}
```

Metoda *Tabel.Increm()* se poate confrunta cu situaţia în care unul dintre parametrii săi (indice) primeşte o valoare necorespunzătoare. În maniera clasică de tratare a erorilor se poate adopta una dintre următoarele rezolvări:

1. ***terminarea forţată a programului*** - nu reprezintă o tratare propriu-zisă a erorii; de multe ori o eroare nu este chiar atât de gravă, iar programul poate şi trebuie să se descurce cu ea într-un mod mai inteligent.
2. ***returnarea unei valori speciale pe post de „eroare“***.
3. ***poziţionarea unui indicator de eroare şi returnarea unei valori valide***, care însă poate implica coruperea stării programului.
4. ***apelul unei funcţii prevăzute special pentru cazurile de eroare***; această funcţie, la rândul ei are de ales între cele 3 alternative de mai sus.

În exemplul prezentat, metoda *indexOf()* aplică varianta 2), iar metoda *Tabel.Increm()* — varianta 3). Cu privire la varianta 2) trebuie spus ca ea nu este întotdeauna uşor de aplicat. Dacă pentru funcţia *indexOf()*, de exemplu, valoarea -1 ca indicator de eşec este satisfăcătoare (este clar că un indice de tablou nu poate fi negativ în mod normal), în schimb, pentru *Increm()* se pune întrebarea ce valoare a tipului *int* ar putea fi considerată ca „ieşită din comun“? Pentru

asemenea funcţii o soluţie ar putea fi aceea că, în loc de *int*, tipul returnat să fie o structură compusă din rezultatul propriu-zis şi un fanion de eroare.

Pe de altă parte, chiar şi acolo unde este aplicabilă, opţiunea 2) implică necesitatea testării de fiecare dată a rezultatului execuţiei funcţiilor respective. Acest lucru poate duce uşor la dublarea dimensiunii codului. Alternativa 3) prezintă dezavantajul că apelantul funcţiei în care a apărut eroarea poate să nu „observe“ că s-a întâmplat ceva. De exemplu, multe funcţii din biblioteca standard C indică apariţia unei erori prin setarea variabilei globale *errno*. Dar prea puţine programe ale utilizatorilor testează valoarea lui *errno* suficient de sistematic încât să se obţină o tratare consistentă a erorilor.

În concluzie, metodele clasice de tratare a excepţiilor se caracterizează prin faptul că porţiunile de cod „normale“ se întrepătrund cu cele de tratare a excepţiilor, iar claritatea programelor are de suferit din cauza ramificărilor generate de numeroasele teste ale indicatorilor de eroare..

Mecanismul de emitere-captare a excepţiilor

Mecanismul de emitere-captare a excepţiilor are două avantaje esenţiale:

- permite separarea explicită a secvenţelor de cod care tratează situaţiile normale faţă de cele care tratează excepţiile.
- rezolvă „dilema“ de la alternativa 2) de tratare clasică a erorilor, prin aceea că oferă suportul necesar pentru a „forţa“ o funcţie să returneze, în situaţii deosebite, valori ale altui tip decât cel specificat în antet ca tip returnat.

Se va ilustra în secvenţa de mai jos varianta „cu excepţii“ a metodei *Tabel.Increm()* din programul prezentat în paragraful anterior:

```
class IndiceEronat extends Exception{ //clasa pt. reprezentarea exceptiei
        private int lung;
        private int index_er;
        public IndiceEronat(int l,int i) {//constructor
            lung=l; index_er=i;
        }
        public String toString(){
            return "Indicele "+index_er+" eronat pentru tabloul de lungime "+ lung;
        }
}

class Tabel{
        public static int Increm(int[ ] tab,int indice,int val) throws IndiceEronat{
            if(indice>=0 && indice<tab.length){
                tab[indice]+=val;
                return tab[indice];
            }
            else //emitere exceptie
                throw new IndiceEronat(tab.length, indice);
       }        //. . .
}
```

```
class ClientTabel{
        public static void main(String[ ] arg) {
            int[ ] tab = new int[Integer.parseInt(arg[0])];
            //se completeaza tabloul cu valori
            int suma=0,indice,val;
            boolean gata=false;
            do{
                //se citesc de la tastatura un indice si o valoare
                try { //secventa sensibila la aparitia unei exceptii
                    suma+=Tabel.Increm(tab,indice,val);
                }
                catch(IndiceEronat e){ //tratare exceptie
                    System.out.println("Exceptie IndiceEronat: "+e);
                }
        //. . .
            }while(!gata);
        }
}
```

Ideea fundamentală care stă la baza mecanismului de emitere-captare a excepţiilor este aceea că dacă o funcţie oarecare *F1* se confruntă cu o situaţie pe care nu o poate rezolva, ea va anunţa acest lucru apelantului ei, prin emiterea unei aşa-numite excepţii, „sperând“ că apelantul va putea (direct sau indirect) să rezolve problema. O funcţie *F2* care doreşte să rezolve o situaţie de excepţie trebuie să-si manifeste intenţia de a capta excepţia emisă în situaţia respectivă. În exemplul de mai sus funcţia *F1* este *Tabel.Increm()*, iar *F2* este *ClientTabel.main()*.

Limbajul Java pune la dispoziţie trei construcţii de bază pentru lucrul cu excepţii:

- instrucţiunea ***throw*** pentru emiterea excepţiilor;
- blocul ***catch*** pentru definirea secvenţei de captare şi tratare a unei excepţii; o asemenea secvenţă se mai numeşte *handler* de excepţii;
- blocul ***try*** pentru delimitarea secvenţelor de cod sensibile la apariţia excepţiilor..

Instrucţiunea *throw*

Ca sintaxă, această instrucţiune este asemănătoare cu *return*:

```
throw expresie_exceptie;
```

unde *expresie_exceptie* are ca rezultat o referinţă la un obiect al unei clase derivate (direct sau indirect) din *Throwable*. De obicei, această expresie constă din aplicarea operatorului *new* pentru a se crea un obiect al clasei alese pentru reprezentarea excepţiei.

O instrucţiune *throw* poate să apară într-o funcţie numai dacă:

- ea se găseşte în interiorul unui bloc *try-catch* care captează tipul de excepţie generată de expresia din *throw*, sau
- definiţia funcţiei este însoţită de o clauza *throws* în care apare tipul de excepţie respectiv sau

- excepţia generată aparţine claselor *RunTimeException* sau *Error*, sau descendenţilor acestora

Dacă instrucţiunea *throw* nu apare într-un bloc *try*, efectul ei este asemănător unui *return*: se creează o excepţie care va fi „pasată“ spre apelantul funcţiei respective şi se produce revenirea din funcţie; funcţia însă NU va transmite NICI UN rezultat chiar dacă tipul returnat care apare în semnătura funcţiei nu este *void*. De exemplu, dacă funcţia *Tabel.Increm()* execută instrucţiunea *throw*, ea nu va returna nici un rezultat de tip *int*.

Clauza throws

Această clauză se ataşează la antetul unei funcţii şi ea precizează tipurile de excepţii care pot fi generate pe parcursul execuţiei funcţiei respective:

```
tip_returnat nume_functie(lista_param) throws tip_ex1,
tip_ex2,...,tip_exn
```

Dacă în lista de tipuri din clauza *throws* există cel puţin două tipuri, *tip_exi* şi *tip_exj*, derivate dintr-o superclasă comună *tip_exsup*, limbajul permite înlocuirea celor două cu *tip_exsup*. Este adevărat că, până la urmă, toate tipurile din lista *throws* derivă direct sau indirect din *Exception* (se poate spune chiar din *Throwable* sau *Object*), deci, teoretic clauza poate conţine doar această clasă. În practică este, însă, nerecomandat acest lucru, deoarece, aşa cum tip_returnat reprezintă tipul rezultatului returnat de funcţie la o execuţie normală, tot aşa, clauza *throws* oferă informaţii referitoare la situaţiile de excepţie care pot apărea în funcţia respectivă. Dacă în listă se trece doar *Exception*, informaţia este foarte vagă pentru cititorul programului.

Care tipuri de excepţie vor apărea într-o clauză *throws*?

Dacă utilizatorul îşi defineşte propriile tipuri de excepţie, atunci este foarte probabil ca el să ştie în ce situaţii urmează să se genereze excepţiile respective şi, ca urmare, le va prevedea în clauze *throws* sau în secvenţe *catch*. În cazul excepţiilor predefinite ale limbajului Java, acestea trebuie să apară în clauze *throws* ale funcţiilor care apelează metode din biblioteca Java care, la rândul lor, generează excepţiile respective. Consultând documentaţia claselor predefinite ale limbajului Java se poate constata că la fiecare metodă este indicată, unde e cazul, clauza *throws*. Pentru excepţiile din clasele *RuntimeException* şi *Error*, precum şi din descendenţii acestora nu sunt necesare clauze *throws* (v. exceptiile neverificate).

Dacă utilizatorul nu prevede clauza *throws*, şi nici secvenţe *try-catch* la o funcţie de a sa care apelează funcţii predefinite dotate cu clauze *throws*, compilatorul va sesiza acest lucru ca eroare. Aceasta este una dintre căile prin care utilizatorul poate afla ce ar fi trebuit să pună în clauza *throws* a funcţiei sale. O situaţie frecventă, care constituie exemplu în acest sens este cea a utilizării funcţiilor de intrare/ieşire din pachetul *java.io*, fără a prevedea clauze *throws IOException*.

Blocurile *try-catch*

Din punct de vedere sintactic un bloc *try-catch* are forma de mai jos:

```
try {
    //secventa obisnuita de operatii, care poate fi intrerupta de aparitia unei exceptii
}
catch(tip_ex1 e){
    //tratare exceptie de tipul tip_ex1
}
catch(tip_ex2 e){
    //tratare exceptie de tipul tip_ex2
}
//. . .
catch(tip_exn e){
    //tratare exceptie de tipul tip_exn
}
finally {
    //secventa optionala (despre finally vom vorbi intrun paragraf urmator)
}
//. . .(*)
```

Un bloc *try* este urmat de 0, 1 sau mai multe blocuri *catch* şi, opţional, de un bloc *finally*. Vom presupune deocamdată că nu avem bloc *finally*. Modul de funcţionare al unei asemenea construcţii este următorul:

- dacă în decursul desfăşurării secvenţei de operaţii din blocul *try* este emisă o excepţie, fie direct (adică prin execuţia unui *throw* explicit), fie indirect (adică prin apelul unei funcţii care emite excepţia), secvenţa se întrerupe şi se baleiază lista de blocuri catch asociate pentru a-l detecta pe cel corespunzător tipului de excepţie emisă;
- dacă un astfel de bloc există, se va lansa în execuţie secvenţa de tratare respectivă.
- dacă această secvenţă nu trece printr-un *return* sau un *throw*, la terminarea ei execuţia programului continuă cu linia care urmează după ultimul bloc catch sau, după caz, *finally* ataşate blocului *try* curent (cea notata cu (*) în secvenţa de mai sus).
- tot de la acea linie se reia execuţia şi dacă secvenţa *try* a rulat fără să fi apărut vreo excepţie.

Când spunem bloc catch corespunzător unei excepţii, aceasta înseamnă că tipul specificat ca parametru după cuvântul catch este fie identic cu tipul excepţiei, fie este o superclasă a tipului excepţiei. Dacă nu există nici un bloc catch corespunzător excepţiei emise, aceasta va fi propulsată spre apelanţii funcţiei curente, căutându-se un eventual bloc *try* înconjurător. Practic, un bloc catch poate fi asimilat cu o funcţie anonimă, cu un singur parametru specificat imediat după cuvântul catch.

Apelul unei asemenea funcţii nu se face explicit, ci automat, dacă în secvenţa *try* asociată apare o excepţie corespunzătoare. Un bloc catch se poate termina normal (adică prin epuizarea instrucţiunilor care îl compun) sau printr-un *throw* care să emită o nouă excepţie sau tot pe cea primită. În acest din urmă caz are loc retransmisia excepţiei.

Referitor la drumul parcurs de o excepţie din momentul emiterii sale şi până ajunge să fie tratată, lucrurile stau astfel:

- stiva de apeluri se baleiază dinspre vârf spre bază în căutarea unui bloc *try* înconjurător în raport cu instrucţiunea *throw* emitentă. Un bloc *try* este înconjurător faţă de o instrucţiune *throw* dacă aceasta se găseşte chiar în interiorul acelui bloc *try* sau dacă se află într-o funcţie apelată (direct sau indirect) din acel bloc *try*.

- dacă se găseşte un bloc *try* înconjurător, se caută între blocurile catch ataşate, unul ce corespunde tipului excepţiei emise. Dacă există un astfel de bloc catch, acesta preia excepţia. În caz contrar, excepţia este propagată mai departe, repetându-se procesul de la pasul anterior. Dacă se ajunge cu căutarea până în funcţia *main()* şi nici aici nu există un *handler* adecvat, excepţia este preluată de *handler*-ul de excepţii al maşinii virtuale care determină abandonul programului, cu afişarea unui mesaj corespunzător.
- dacă excepţia este preluată de un bloc catch şi acesta nu o retransmite, excepţia respectivă se consideră a fi tratată, indiferent de modul în care *handler*-ul operează efectiv (în particular un *handler* poate avea corpul de instrucţiuni vid).

Procesul de baleiere a stivei de apeluri implică, practic, părăsirea tuturor funcţiilor „întâlnite“ în cale până la găsirea blocului *try* înconjurător. Cu ajutorul unui exemplu vom ilustra mecanismul de funcţionare a blocurilor *try-catch* şi a instrucţiunii *throw*:

```
class oExceptie extends Exception{/*...*/}
class altaExceptie extends Exception{/*...*/}

class OClasa {
        public static void oFunctie( ) throws oExceptie {
            //. . .
            if(eroare) throw new oExceptie( );
        }
        public static void altaFunctie( ) throws oExceptie, altaExceptie {
            oFunctie( );
        }
        public static void siIncaUna ( ) throws oExceptie,altaExeceptie{
        //. . .
            try{ // 1
                  altaFunctie( );
            }
            catch(oExceptie e){
                if(poti_sa_rezolvi) fa_o( );
                else throw e; // retransmisia exceptiei e
            }
            catch(altaExceptie e){
                if(alta_eroare) throw new altaExceptie();
                else tratare_normala( );
            }
            try{ // 2
                //presupunem ca aici nu se apeleaza oFunctie( ) si nici altaFunctie( )
            }
            catch(Exception e){
                //capteaza orice exceptie derivata din Exception aparuta in try 2
            }
            //instructiunea imediat urmatoare blocului trycatch 2
        }//end siIncaUna()
        public static void main(String[ ] arg) {
```

```
            //. . .
        }
        //. . .
}//end class OClasa
```

Presupunem că la un moment dat lanţul de apeluri este:

```
main() -> siIncaUna() -> altaFunctie() -> oFunctie()
```

şi că *oFunctie()* execută instrucţiunea *throw*. Se creează o excepţie de tip *oExceptie* şi se iese din *oFunctie(),* revenindu-se în *altaFunctie()*. Aici se constată că nu există bloc *try* şi, ca urmare, se iese din *altaFunctie()* ca şi cum aici ar exista un *throw*, şi se revine în *siIncaUna()*. De data aceasta revenirea are loc în interiorul unui bloc *try* (cel notat cu //1). Se caută printre ramurile sale catch şi se găseşte cea cu parametru de tip *oExceptie*. Dacă excepţia se poate rezolva, după execuţia secvenţei catch, programul continuă cu blocul *try* următor (cel notat cu //2). Dacă se ajunge la retransmisia excepţiei, atunci se iese din *siIncaUna()* şi se verifică dacă apelul la această funcţie nu se află în interiorul unui bloc *try* din apelant (funcţia *main()*, adică).

La execuţia unui *throw* din interiorul unui bloc catch nu se rebaleiază lista curentă de *handler*-e şi nici nu se trece la următorul bloc *try-catch* în aceeaşi funcţie, chiar dacă acesta din urmă ar putea capta excepţia emisă (cum ar fi cazul blocului *try* //2 din exemplul de mai sus). Există şi posibilitatea ca un bloc *try-catch* să fie inclus în secvenţa *try* a unui alt bloc *try-catch*, ca în exemplul următor:

```
class oExceptie extends Exception{/*...*/}
class altaExceptie extends Exception{/*...*/}
class OClasa {
        public static void oFunctie( ) throws oExceptie{
            //. . .
            if(eroare) throw new oExceptie( );
        }
        public static void altaFunctie( ) throws oExceptie, altaExceptie{
            try { //1
                try { //2
                    oFunctie( );
                }
                catch (altaExceptie e){
                     System.out.println("S-a captat  altaExceptie"); //aici tratarea
                    //presupune pur si simplu o afisare de mesaj
                }
            }
            catch(oExceptie e){
                 if(poti_sa_rezolvi) fa_o( );
                 else throw e; // retransmisia exceptiei e
            }
            catch(altaExceptie e){
                 if(alta_eroare) throw new altaExceptie();
                 else tratare_normala( );
```

```
            }
        }
}
public static void main(String[ ] arg) {
        //. . .
}
//. . .
}//end class OClasa
```

În acest exemplu se observă că blocul *try-catch* notat cu //2 nu va capta excepţii de tipul *oExceptie*. Blocul *try-catch* notat cu //1 este considerat ca înconjurător pentru blocul //2 şi, deci, excepţia emisă de *oFunctie()* va ajunge să fie tratată de *handler-ul* corespunzător ataşat blocului //1. Astfel, un bloc *try* este înconjurător pentru un *throw* dacă secvenţa *try*

- conţine în mod direct acel *throw*, sau
- conţine un bloc *try-catch* în care se află acel *throw*, sau
- conţine rădăcina apelului la funcţia în care se află acel *throw*.

Secvenţa finally

Dacă este prezentă, se execută întotdeauna după ce s-a terminat secvenţa *try* sau secvenţele catch din blocul *try-catch* curent. Practic, nu există posibilitatea de a determina desfăşurarea execuţiei unui bloc *try* prevăzut cu ramura *finally* în sensul „şuntării“ acestei secvenţe. Secvenţa *finally* reprezintă un mecanism prin care se forţează execuţia unei porţiuni de cod indiferent dacă o excepţie a apărut sau nu. De obicei, ramura *finally* are rolul de a „face curat“, în sensul de a elibera resurse de genul fişiere deschise accesate prin intermediul unor variabile locale.

```
public boolean Cautare(String nume_fis,String cuvant ) throws IOException{
        BufferedReader flux_in = null;
        try{
            flux_in = new (new InputStreamReader(new FileInputStream(nume_fis)));
            String linie;
             while ((linie = flux_in.readLine()) != null)
                  if(linie.equals(cuvant)) return true;
                        return false; //nu s-a gasit cuvantul in fisier
        }
        finally {
             if(flux_in != null) flux_in.close();
}
```

Primul lucru pe care îl constatăm la programul prezentat este absenţa ramurii *catch*. Aceasta înseamnă că programatorul nu are intenţia să trateze excepţiile de tip *IOException* în cadrul funcţiei *Cautare()*. Mai mult, acest program este un exemplu de utilizare a blocului *try* nu neapărat pentru a rezolva situaţii de eroare, ci, datorită blocului *finally*, mai mult pentru a determina o anumită înlănţuire a operaţiilor. De altfel, trebuie să ne obişnuim cu ideea că o excepţie nu este întotdeauna un indiciu de eroare, ci, uneori, este pur şi simplu un semn că în desfăşurarea programului a intervenit ceva deosebit care trebuie tratat altfel.

În secvenţa de mai sus dacă fişierul nu ar fi putut fi deschis (de exemplu *nume_fis* nu există) atunci referinţa flux_in ar fi rămas pe *null*. De aceea s-a prevăzut testul din blocul *finally*. Presupunând că deschiderea fişierului ar fi mers cum trebuie, în continuare, indiferent de modul în care se termină blocul *try* (pe unul dintre cele două return-uri sau prin apariţia unei excepţii generate de funcţia *readLine()*) prezenţa ramurii *finally* asigură închiderea fişierului înainte de părăsirea efectivă a funcţiei *Cautare()*. Ramura *finally* se mai poate utiliza şi pentru a forţa execuţia unei porţiuni de cod când în *try* avem bucle ce conţin combinaţii de *break*, continue şi *return*.

Excepţii predefinite ale limbajului Java

Există două clase principale de excepţii predefinite: *Error* şi *Exception*. Excepţiile reprezentate de clasele descendente din *Error* indică, în general, probleme foarte serioase care trebuie să ducă la întreruperea programului şi nu se recomandă captarea lor de către utilizator. Ele sunt generate automat de suportul de execuţie al programelor Java, la întâlnirea situaţiilor de eroare respective. Clasele *Error* şi *RuntimeException*, împreună cu descendenţii lor, formează categoria excepţiilor neverificate (*unchecked*), adică excepţii care pot fi generate în programe, fără obligativitatea ca ele să apară în clauze *throws*.

Restul excepţiilor sunt verificate (*checked*), adică la compilare se verifică dacă există clauze *throws* corespunzătoare. Un exemplu în acest sens este clasa *IOException* (definită în pachetul *java.io*), care trebuie specificată fie în clauze *throws*, fie în ramuri catch, acolo unde se folosesc funcţiile de intrare/ieşire din pachetul *java.io*.

Programatorul poate să-şi definească anumite clase de excepţii ca descendente din *RuntimeException*, făcându-le, astfel, neverificate. Acest lucru trebuie evitat, deoarece lipsa clauzei *throws* din declaraţia unei funcţii care totuşi generează excepţii, este, de fapt, o lipsă de informaţie pentru potenţialii utilizatori ai funcţiei respective.

În exemplul care urmează vom arata cum putem folosi mecanismul excepţiilor pentru a rezolva problema depăşirii limitelor unui tablou. Practic, vom completa elementele unui tablou fără ca, la parcurgerea tabloului, să aplicăm comparaţia clasică a indicelui faţă de lungimea tabloului (*i < tab.length*). Aici ne vom folosi de o clasă de excepţii predefinită, *IndexOutOfBoundsException*, din pachetul *java.lang*:

```
import java.io.*;

public class Tablou {
        public static void main (String[ ] args) {
            int [ ] tab = new int[10];
            int i;
            try {
                 for(i=0;;) tab[i++]=i; //parcurg tabloul fara grija
            }
            catch(IndexOutOfBoundsException e) {
                  //ajung aici cand i >=tab.length
                 System.out.println(e);
                 System.out.println("Gata tabloul!");
            }
        }
}
```

Se observă că aici nu avem instrucţiune *throw*, adică nu avem o emitere explicită a excepţiei. Aceasta, deoarece în anumite situaţii, cum este şi depăşirea limitelor tablourilor, excepţiile sunt generate automat, de către suportul de execuţie Java. Excepţia *IndexOfBoundException* este derivată din *RuntimeException*, deci este neverificată. Aceasta înseamnă că şi dacă în programul de mai sus nu am fi prevăzut ramura *catch* pentru ea, nu am fi fost obligaţi să ataşăm funcţiei *main()* o clauză *throws IndexOutOfBoundsException*.

Temă

1. Să se scrie un program care citeşte de la tastatură o pereche de numere în care primul număr trebuie să fie mai mic decât al doilea. Dacă această condiţie nu este îndeplinită, folosind mecanismul excepţiilor, se va semnala eroare şi se va trata această eroare prin cererea unei alte perechi de numere. Perechea care îndeplineşte condiţia va fi scrisă într-un fişier primit ca parametru la apel.
2. Să se scrie un program care citeşte două numere de la tastatură şi afişează rezultatul împărţirii celor două numere. Programul va utiliza mecanismul de tratare a excepţiilor pentru a cere reintroducerea împărţitorului atunci când se realizează o împărţire la 0, şi pentru a cere reintroducerea corectă a numerelor dacă în loc de numere s-au introdus cuvinte.
3. Să se scrie un program care citeşte numele şi CNP-ul unei persoane. Mecanismul de tratare a excepţiilor va fi utilizat pentru introducerea unui CNP corect. Dacă CNP-ul a fost introdus corect va fi utilizat pentru a determina data naşterii şi a calcula vârsta.