Lecţia 8

Tratarea excepţiilor

Un program comercial, fie el scris în Java sau în orice alt limbaj de programare, trebuie să ţină cont de posibilitatea apariţiei la execuţie a unor anumite situaţii neobişnuite: fişierul din care se doreşte a se citi o informaţie nu există, utilizatorul a introdus un şir de caractere de la tastatură dar care nu reprezintă un număr aşa cum a cerut programul, ş.a.m.d. În această lucrare vom studia un mecanism dedicat tratării acestor situaţii excepţionale.

8.1 Excepţiile şi tratarea lor

8.1.1 Ce este o excepţie?

Ideal, toate situaţiile neobişnuite ce pot să apară pe parcursul execuţiei unui program trebuie detectate şi tratate corespunzător de acesta. Motivul? Există o sumedenie. De exemplu, un program ar putea cere utilizatorului să introducă de la tastatură un număr, dar acesta să intoducă un şir de caractere ce nu reprezintă un număr. Un program bine gândit va detecta această problemă, va anunţa greşeala şi probabil va cere utilizatorului reintroducerea numărului. Un program care se opreşte subit în momentul în care utilizatorul face o greşeală, fără nici un avertisment şi fără nici un mesaj clar care să anunţe problema, va "băga în ceaţă" orice utilizator.

Exemplul de mai sus este doar un caz particular de situaţie neobişnuită sau de excepţie. Situaţiile neobişnuite pot apare pe parcursul execuţiei programului şi din motive independente de utilizator. De exemplu, am putea avea într-un program o metodă care trebuie să tipărească pe ecran elementul i al unui tablou de întregi, valoarea lui i şi tabloul fiind specificaţi prin parametrii metodei. Dacă valoarea indexului depăşeşte limitele tabloului, vorbim tot de o situaţie neobişnuită, deşi ea are un iz de eroare de programare. Şi aceste situaţii ar trebui detectate de un program chiar dacă utilizatorul nu prea are ce face în cazul unei erori de programare. Tratarea unei astfel de situaţii ar putea implica de exemplu crearea unui raport de eroare de către utilizator la cererea

programului. Tot este mai bine decât oprirea subită a programului.

Atenţie În general, o eroare este o excepţie, dar o excepţie nu reprezintă neapărat o eroare de programare.

Din exemplele de până acum s-ar putea crede că tratarea unei excepţii implică într-o formă sau alta intervenţia utilizatorului. Nu este deloc aşa. O practică în industria aero-spaţială (e drept destul de primitivă) este ca o aceeaşi parte dintr-un program să fie scrisă de două ori de programatori diferiţi. Dacă la execuţie programul de control al sistemului de zbor detectează o eroare de programare într-un exemplar al respectivei părţi, el va cere automat celui de-al doilea exemplar să preia responsabilităţile primului exemplar. Acesta este doar un exemplu în care chiar şi unele erori de programare pot fi tratate de un program. Să nu mai vorbim de situaţii mai simple. De exemplu, pentru a îndeplini anumite funcţionalităţi un program trebuie să acceseze serviciile puse la dispoziţie de un server. Se poate întâmpla ca la un moment dat respectivul server să fie suprasolicitat şi să nu poată răspunde la cererea programului. Şi o astfel de situaţie neobişnuită poate fi văzută ca o excepţie la apariţia căreia programul ar putea încerca, de exemplu, să acceseze automat un alt server care pune la dispoziţie aceleaşi servicii.

În urma acestei discuţii putem defini o excepţie în felul următor:

Definiţie 1 *O excepţie reprezintă un eveniment deosebit ce poate apare pe parcursul execuţiei unui program şi care necesită o deviere de la cursul normal de execuţie al programului.*

8.1.2 Metode clasice de tratare a excepţiilor

Pentru a ilustra modul clasic de tratare a situaţiilor neobişnuite şi a deficienţelor sale vom considera un exemplu ipotetic. Să presupunem un program care manipulează şiruri de numere reale. Clasa de mai jos modelează un astfel de şir. Pentru simplitate vom considera că asupra unui şir putem efectua două operaţii: adăugarea unui număr la sfârşitul său, respectiv extragerea din şir al primului număr de după prima apariţie în şir a unui număr dat. O cerinţă suplimentară asupra acestei clase este ca diferite situaţii anormale să poată fi puse în evidenţă de apelanţii metodelor ei.

```
class SirNumereReale {

    private double[] sir = new double[100];
    private int nr = 0;
    private boolean exceptie = false;

    public boolean testExceptie() {
        return exceptie;
```

```
    }

    public boolean adauga(double x) {
        if(nr == sir.length) {
            return false;
        }
        sir[nr] = x;
        nr++;
        return true;
    }

    public double extrageDupa(double x) {
        int i;
        for(i = 0; i < nr; i++) {
            if(sir[i] == x) {
                if(i + 1 < nr) {
                    double result = sir[i + 1];
                    for(int j = i + 1; j < nr - 1; j++) {
                        sir[j] = sir[j + 1];
                    }
                    nr--;
                    exceptie = false;
                    return result;
                } else {
                    exceptie = true;
                    return -2;
                }
            }
        }
        exceptie = true;
        return -1;
    }
}
```

Un lucru interesant de observat este că situaţiile neobişnuite sunt tratate utilizând convenţii. Astfel, apelantul metodei de adăugare îşi va putea da seama că operaţia a eşuat testând valoarea returnată de metodă. În cazul celei de-a doua metode lucrurile sunt mai complicate. Este nevoie de un fanion care să fie testat după fiecare apel la metoda de extragere. Dacă el indică faptul că avem o situaţie neobişnuită atunci, pe baza valorii returnate de metodă, apelantul metodei îşi poate da seama ce s-a întâmplat: valoarea -1 spune apelantului că în acel şir nu există numărul dat ca parametru, iar valoarea -2 îi spune că după numărul dat ca parametru nu mai există nici un număr. Necesitatea fanionului e evidentă: valoarea -1 poate reprezenta chiar un număr din şir. Distincţia între valoarea -1 ca număr din şir şi valoarea -1 ca şi cod de identificare a problemei apărute este realizată prin intermediul fanionului.

La prima vedere, aceste convenţii par bune. Din păcate, ele fac utilizarea unui obiect şir destul de dificilă. Să considerăm acum că trebuie să scriem o metodă care primeşte ca parametru un număr şi un şir şi trebuie să extragă primele 10 numere ce apar în şir după prima apariţie a numărului dat ca parametru şi să le calculeze media. Dacă nu există 10 numere după numărul dat se va returna 0. Este important de remarcat că aceasta nu e o convenţie de codificare a unei situaţii neobişnuite ci doar o cerinţă pe care trebuie să o implementeze metoda. Codul metodei e prezentat mai jos.

```
class Utilitar {

    public static double medie(double x, SirNumereReale sir) {
        double medie = 0;
        for(int i = 0; i < 10; i++) {
            double tmp = sir.extrageDupa(x);
            if(!sir.testExceptie()) {
                medie+= tmp;
            } else {
                medie = 0;
                break;
            }
        }
        return medie / 10;
    }
}
```

La prima vedere, acest cod este corect. Din păcate nu este aşa şi codul totuşi e compilabil! Ce se întâmplă dacă parametrul x nu există în şirul dat? Evident, undeva în sistem această problemă va fi sesizată sub forma unei erori de programare. Unde? Depinde. Poate în apelantul metodei *medie*, poate în apelantul apelantului metodei *medie*, poate la un milion de apeluri distanţă. Cine este de vină? Apelantul metodei *medie* va da vina pe cel care a implementat metoda *medie* deoarece el trebuia să returneze 0 doar dacă nu existau 10 numere în şir după parametrul x. Are dreptate. Programatorul ce a implementat metoda *medie* dă vina pe specificaţiile primite care nu spuneau nimic de faptul că parametrul x ar putea să nu apară în şir. Are dreptate. Managerul de proiect dă vina însă tot pe el pentru că nu a sesizat această inconsistenţă şi nu a anunţat-o. Programatorul replică prin faptul că a considerat că metoda se apelează totdeauna cu un şir ce-l conţine pe x. După îndelungi vociferări, în urma cărora tot se va găsi un vinovat, codul va fi modificat ca mai jos.

```
class Utilitar {

    private static notFound = false;

    public static boolean elementLipsa() {
```

```
        return notFound;
    }

    public static double medie(double x, SirNumereReale sir) {
        double medie = sir.extrageDupa(x);
        notFound = false;
        if(sir.testExceptie()) {
            if(medie == -1) {
                notFound = true;
                return 0; //desi nu inseamna nimic
            } else {
                notFound = false;
                return 0;
            }
        }
        for(int i = 1; i < 10; i++) {
            double tmp = sir.extrageDupa(x);
            if(!sir.testExceptie()) {
                medie+= tmp;
            } else {
                medie = 0;
                break;
            }
        }
        return medie / 10;
    }
}
```

După cum se poate vedea din exemplul de mai sus, tratarea clasică a excepţiilor ridică o serie de probleme. Pe de-o parte codul poate deveni destul de greu de urmărit datorită nenumăratelor teste care trebuie realizate şi care se întreţes în codul esenţial. Cea mai mare parte din codul metodei *medie* testează valoarea fanioanelor pentru a detecta situaţii neobişnuite şi pentru a le trata corespunzător. Aceste teste se întrepătrund cu ceea ce face metoda de fapt: ia 10 numere din şir ce apar după parametrul x şi le calculează media. Efectul este că metoda va fi foarte greu de înţeles pentru o persoană care vede codul prima dată. Pe de altă parte, cea mai mare problemă a modului clasic de tratare a excepţiilor este că el se bazează pe convenţii: după fiecare apel al metodei *extragDupa* trebuie verificat fanionul pentru a vedea dacă nu a apărut vreo situaţie neobişnuită. Din păcate convenţiile se încalcă iar compilatorul nu poate să verifice respectarea lor.

Revenind la exemplul nostru, mai exact la programatorul ce a implementat metoda *medie*, el poate aduce un argument incontestabil în apărarea sa: dacă programul este scris în Java, atunci programatorul ce a implementat clasa *SirNumereReale* de ce nu a folosit mecanismul excepţiilor verificate pentru a anunţa situaţiile neobişnuite?

8.2 Tratarea excepţiilor în Java

8.2.1 Definirea excepţiilor

După cum am văzut deja, în Java aproape orice este văzut ca un obiect. Prin urmare nu ar trebui să ne mire faptul că o excepţie nu este altceva decât un obiect care se defineşte aproape la fel ca orice alt obiect. Singura condiţie suplimentară ce trebuie satisfăcută este să fie moştenită direct ori indirect clasa predefinită *Throwable* de către clasa obiectului nostru excepţie. În practică, la definirea excepţiilor NU se extinde însă această clasă, ci se utilizează clasa *Exception*, o subclasă a clasei *Throwable*. Să vedem acum cum definim excepţiile pentru situaţiile speciale din exemplul de la începutul lecţiei.

```
class ExceptieSirPlin extends Exception {

    public ExceptieSirPlin(String mesaj) {
        super(mesaj);
    }
}

class ExceptieNumarAbsent extends Exception {

    private double nr;

    public ExceptieNumarAbsent(double nr) {
        super(''Numarul '' + nr + '' nu apare in sir!'');
        this.nr = nr;
    }

    public int getNumarAbsent() {
        return nr;
    }
}

class ExceptieNuExistaNumere extends Exception {

    public ExceptieNuExistaNumere() {
        super(''Nu mai exista numere dupa numarul dat!'');
    }
}
```

8.2.2 Clauza throws

Încă din prima lecţie am spus că obiectele ce compun un sistem software interacţionează prin apeluri de metode. Din perspectiva unui obiect care apelează o metodă a unui alt obiect, la revenirea din respectivul apel putem fi în două situaţii: metoda s-a executat în

mod obişnuit sau metoda s-a terminat în mod neobişnuit datorită unei excepţii. Clauza *throws* apare în antetul unei metode şi ne spune ce tipuri de excepţii pot conduce la **terminarea** neobişnuită a respectivei metode. Mai jos prezentăm modul în care specificăm faptul că metodele clasei *SirNumereReale* pot să se termine datorită unor situaţii neobişnuite şi care sunt acele situaţii.

```
class SirNumereReale {

    //Implementarea clasei nu conteaza pentru client.
    //El cunoaste doar interfata obiectului data de metodele
    //publice ale clasei.

    public void adauga(double x) throws ExceptieSirPlin {
        ...
    }

    public double extrageDupa(double x)
                  throws ExceptieNumarAbsent, ExceptieNuExistaNumere {
        ...
    }
}
```

Clauza *throws* poate fi văzută ca o specificare suplimentară a tipului returnat de o metodă. De exemplu, metoda *extrageDupa* returnează în mod obişnuit o valoare *double* reprezentând un număr din şir. Clauza *throws spune* că în situaţii excepţionale, metoda poate returna o referinţă la un obiect *ExceptieNumarAbsent*, o referinţă la un obiect *ExceptieNuExistaNumere* sau o referinţă la un obiect a cărui clasă (tip) moşteneşte clasele (tipurile) *ExceptieNumarAbsent* sau *ExceptieNuExistaNumere*.

Atenţie Faptul că în clauza *throws* a unei metode apare un tip A înseamă că respectiva metodă se poate termina fie cu o excepţie de tip A, fie cu orice excepţie a cărei clasă moşteneşte direct ori indirect clasa A.

8.2.3 Interceptarea excepţiilor

Până acum am vorbit despre modul de definire a unei excepţii şi de modul de specificare a faptului că o metodă poate să se termine cu excepţie. Dar cum poate afla un obiect ce apelează o metodă ce se poate termina cu excepţie dacă metoda s-a terminat normal sau nu? Răspunsul e simplu: dacă îl interesează acest lucru trebuie să utilizeze un bloc *try-catch-finally*. Structura generală a unui astfel de bloc este prezentată mai jos.

```
try {
    //Secventa de instructiuni ce trateaza situatia normala
    //de executie, dar in care ar putea apare exceptii
```

```
} catch(TipExceptie1 e) {
    //Secventa de instructiuni ce se executa cand in sectiunea
    //try apare o exceptie de tip TipExceptie1 sau de un subtip de-al sau
    //Parametrul e o referinta la exceptia prinsa
} catch(TipExceptie2 e) {
    //Secventa de instructiuni ce se executa cand in sectiunea
    //try apare o exceptie de tip TipExceptie2 sau de un subtip de-al sau
    //Parametrul e o referinta la exceptia prinsa
}
//... Pot apare 0 sau mai multe sectiuni catch
} cacth(TipExceptieN e) {
    //Secventa de instructiuni ce se executa cand in sectiunea
    //try apare o exceptie de tip TipExceptieN sau de un subtip de-al sau
    //Parametrul e o referinta la exceptia prinsa
} finally {
    //Secventa de instructiuni ce se executa in orice conditii la
    //terminarea executiei celorlalte sectiuni
    //Sectiunea finally e optionala si e numai una
}
//aici urmeaza alte instructiuni (*)
```

Din punct de vedere static, secţiunile blocului *try-catch-finally* au următoarea utilizare. În secţiunea *try* se introduce codul pe parcursul căruia pot apare excepţii. În fiecare secţiune *catch* se amplasează secvenţa de instrucţiuni ce trebuie să se execute în momentul în care în secţiunea *try* a apărut o excepţie de tipul parametrului secţiunii *catch* sau de un subtip de-al său. În secţiunea *finally* se amplasează cod ce trebuie neapărat să se execute înaintea părăsirii blocului *try-catch-finally*, indiferent de faptul că în secţiunea *try* a apărut sau nu vreo excepţie şi indiferent dacă ea a fost interceptată ori nu de vreo secţiune *catch*.

Cum funcţionează la execuţie un bloc *try-catch-finally*? În situaţia în care nu apare nici o excepţie în secţiunea *try* lucrurile sunt simple: se execută secţiunea *try* ca orice alt bloc de instrucţiuni, apoi se execută secţiunea *finally* dacă ea există şi apoi se continuă cu prima instrucţiune de după blocul *try-catch-finally*. Prin urmare secţiunile *catch* NU se execută dacă nu apare vreo excepţie în secţiunea *try*.

În continuare, pentru a înţelege cum funcţionează un bloc *try-catch-finally* atunci când apar excepţii, trebuie să înţelegem ce se întâmplă la execuţie în momentul apariţiei unei excepţii. În primul rând trebuie să spunem că la execuţia programului, dacă se emite o excepţie, execuţia "normală" a programului e întreruptă până în momentul în care excepţia respectivă este tratată (interceptată). Prinderea unei excepţii înseamnă executarea unei secţiuni *catch* dintr-un bloc *try-catch-finally*, secţiune *catch* asociată acelei excepţii. Din momentul emiterii şi până în momentul prinderii se aplică în mod repetat un algoritm pe care-l vom prezenta în continuare. Algoritmul se aplică mai

întâi pentru metoda în care a apărut excepţia. Dacă în urma aplicării algoritmului se constată că excepţia nu e tratată în acea metodă, algoritmul se repetă pentru metoda ce a apelat metoda curentă. Acest lucru se tot repetă până se ajunge la o metodă ce prinde excepţia. Dacă mergând înapoi pe stiva de apeluri se ajunge până în metoda *main* şi nici aici nu este prinsă excepţia, maşina virtuală Java va afişa pe ecran mesajul excepţiei, şirul de metode prin care s-a trecut, după care va opri programul.

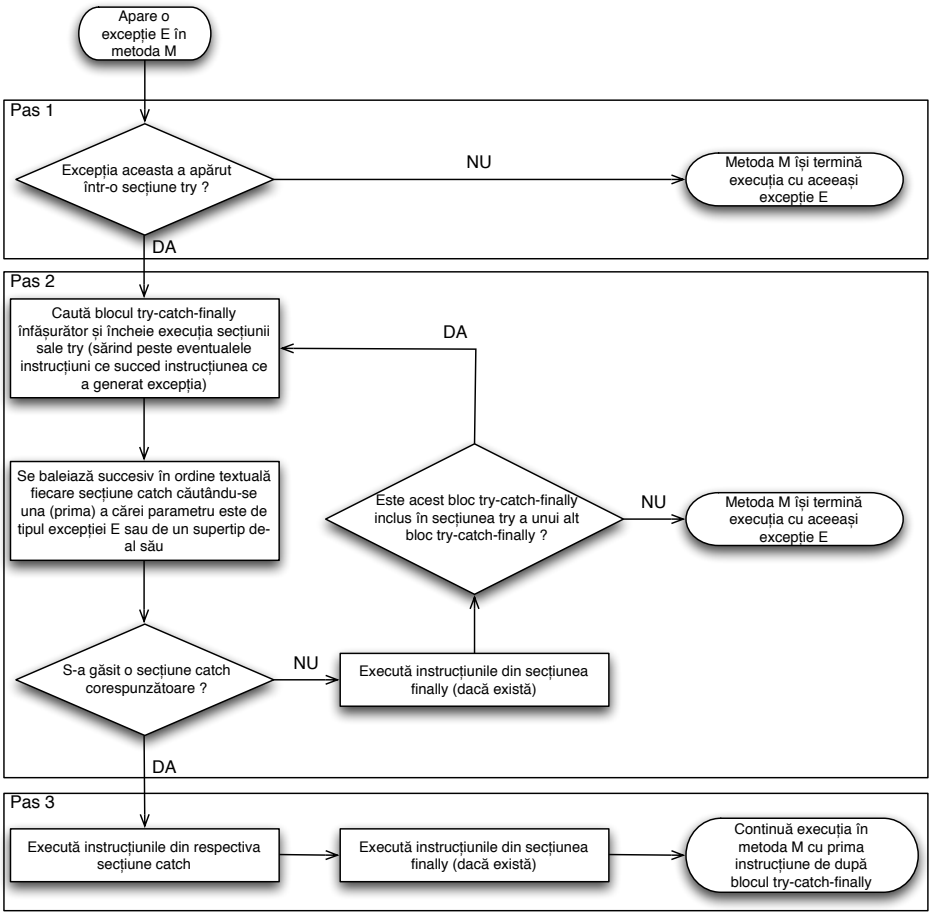


Figura 8.1: Cum se decide dacă metoda M prinde excepţia E sau se termină şi ea cu aceeaşi excepţie E.

Să vedem acum algoritmul despre care am vorbit mai sus. El se aplică unei metode în momentul în care apare una din situaţiile de mai jos:

- s-a executat o instrucţiune care a emis explicit ori implicit o excepţie.
- s-a executat un apel la o metodă iar ea s-a terminat cu o excepţie.

În primul pas al algoritmului se verifică dacă respectiva instrucţiune/apel apare într-o secţiune *try* a unui bloc *try-catch-finally*. Dacă nu, respectiva metodă se termină automat cu aceeaşi excepţie deoarece ea nu interceptează respectiva excepţie. Altfel se trece la pasul doi.

În al doilea pas se caută cel mai apropiat bloc *try-catch-finally* (blocurile *try-catch-finally* pot fi încuibate în sensul că un bloc apare în secţiunea *try* a unui bloc înfăşurător). Odată găsit, algoritmul baleiază în ordine textuală fiecare secţiune *catch* şi verifică dacă tipul concret al excepţiei apărute este de tipul parametrului sau de un subtip de-al tipului parametrului. La găsirea primei clauze *catch* ce respectă această regulă căutarea se opreşte şi se trece la pasul trei. Dacă nu se găseşte o astfel de secţiune *catch* se trece automat la execuţia secţiunii *finally* (dacă ea există) după care se repetă pasul doi pentru un eventual bloc *try-catch-finally* a cărui secţiune *try* include blocul *try-catch-finally* curent. Dacă nu există un astfel de bloc metoda noastră se termină cu aceeaşi excepţie ce a declanşat algoritmul deoarece ea nu o interceptează.

În al treilea pas se poate spune cu certitudine că excepţia a fost interceptată (deci algoritmul nu se mai aplică pentru apelantul acestei metode) şi se trece la executarea instrucţiunilor din clauza *catch* ce a prins excepţia. Apoi se execută secţiunea *finally* (dacă ea există) după care se reîncepe execuţia normală a programului de la prima instrucţiune de după blocul *try-catch-finally* ce a interceptat excepţia.

Dacă aţi citit cu atenţie veţi constata că algoritmul nu spune nimic despre ce se întâmplă dacă se emite o excepţie într-o secţiune *catch* sau *finally*. Lucrurile sunt simple: se aplică algoritmul ca pentru orice emitere de excepţie. Situaţia mai puţin clară este atunci când se execută o secţiune *finally* şi excepţia care a declanşat algoritmul nu a fost încă tratată. Într-o astfel de situaţie, excepţia curentă se consideră tratată şi se reîncepe algoritmul pentru excepţia nou emisă. Atenţie însă la faptul că aceste excepţii nu apar în secţiunea *try* a blocului *try-catch-finally* curent!!! Adică o excepţie emisă într-o secţiune *catch* sau *finally* a unui bloc *try-catch-finally* nu poate fi prinsă de o secţiune *catch* a aceluiaşi bloc *try-catch-finally*!

Atenţie la ordinea de amplasare a clauzelor *catch*!!! Dacă ne uităm la fragmentul de cod de mai jos am putea spune că totul e bine. Nu e aşa. În a doua clauză *catch* nu se ajunge niciodată! Deşi în secţiunea *try* apare o excepţie de tip *oExceptie*, ea va fi prinsă totdeauna de prima secţiune *catch* deoarece *oExceptie* e subtip de-al lui *Exception*. Din fericire, compilatorul sesizează astfel de probleme.

```
class oExceptie extends Exception {}
```

```
try {
    //Aici apare o exceptie de tip oExceptie
} catch(Exception e) {
    ...
} catch(oExceptie e) {
    ...
}
```

Să vedem acum cum va arăta codul metodei *medie* prezentate la începutul lecţiei când folosim excepţii.

```
class Utilitar {

    public static double medie(double x, SirNumereReale sir)
                          throws ExceptieNumarAbsent {
        double medie = 0;
        try {
            for(int i = 0; i < 10; i++) {
                medie+=sir.extrageDupa(x);
            }
            medie = medie / 10;
        } catch(ExceptieNuExistaNumere e) {
            medie = 0;
        }
        return medie;
    }
}
```

Să vedem ce se întâmplă la execuţia acestei metode. Presupunem iniţial că numărul x există în şir şi există 10 numere după el. Prin urmare metoda *extrageDupa* nu se va termina cu exceptie, secţiunea *try* se va executa ca orice alt bloc de instrucţiuni iar după terminarea sa se continuă cu prima instrucţiune de după blocul *try-catch-finally*; adică se va executa instrucţiunea *return* şi metoda noastră se termină normal.
Să vedem acum ce se întâmplă dacă după numărul x mai există în şir doar un număr. Prin urmare, la al doilea apel al lui *extrageDupa* metoda se va termina cu o excepţie de tip *ExceptieNuExistaNumere*. Aplicând algoritmul prezentat anterior se constată că excepţia a apărut într-o secţiune *try*. Se baleiază apoi secţiunile *catch* şi se constată că prima este destinată prinderii excepţiilor de tipul celei emise. Prin urmare se trece la execuţia corpului ei şi *medie* va deveni 0. Apoi se trece la execuţia primei instrucţiuni de după *try-catch-finally* şi se execută instrucţiunea return. Este important de observat că în momentul în care *extrageDupa* s-a terminat cu excepţie NU SE MAI EXECUTĂ NIMIC DIN SECŢIUNEA TRY. Pur şi simplu se abandonează secţiunea *try* indiferent ce instrucţiuni sunt acolo. În cazul nostru se abandonează inclusiv ciclul *for* care încă nu s-a terminat când a apărut excepţia.

Să vedem acum ce se întâmplă dacă nu avem numărul x în şir. Evident, *extrageDupa* se termină cu o excepţie *ExceptieNumarAbsent* la primul apel. Se aplică algoritmul prezentat mai sus şi se constată că această excepţie nu e tratată de blocul *try-catch-finally* înconjurător deoarece nu există o clauză *catch* care să prindă această excepţie. Prin urmare, metoda *medie* se va termina cu aceeaşi excepţie. Acesta e şi motivul pentru care apare tipul acestei excepţii în clauza *throws* a metodei. Dacă nu am pune această clauză compilatorul ar da o eroare de compilare. Motivul e simplu: el vede că metoda *extrageDupa* poate să se termine cu *ExceptieNumarAbsent*, dar metoda *medie* nu o interceptează (nu există o secţiune *catch* corespunzătoare). Prin urmare, compilatorul vede că metoda *medie* s-ar putea termina cu această excepţie şi îi spune programatorului să se hotărască: fie interceptează excepţia în această metodă fie specifică explicit că metoda se poate termina cu *ExceptieNumarAbsent*.

Uitaţi-vă la codul de mai jos. Codul nu e compilabil! Motivul e simplu. Clauza *throws* spune compilatorului că metoda se poate termina cu o exceptie de tipul *oExceptie* sau cu o excepţie de un subtip de-al său. Nimic nu-i poate schimba această părere. Pe de altă parte în metoda *altaMetoda* se prind doar excepţii de tip *altaExceptie*. Acest lucru e corect: pot ajunge astfel de excepţii aici. Altceva e problematic pentru compilator: se prind doar anumite cazuri particulare de excepţii *oExceptie*. Ce se întâmplă cu restul? Ca urmare, compilatorul va semnala o eroare ce va fi rezolvată de programator fie prin adăugarea unei secţiuni *catch* pentru excepţii de tip *oExceptie* după secţiunea deja existentă (mai ştiţi de ce obligatoriu după?) fie se utilizează o clauză *throws* pentru metoda *altaMetoda*.

```
class oExceptie extends Exception {}
class altaExceptie extends oExceptie {}

class Test {

    public static void oMetoda() throws oExceptie {...}

    public static void altaMetoda() {
        try { oMetoda();
        } catch(altaExceptie e) {...}
    }
}
```

Din acest exemplu reies clar avantajele utilizării excepţiilor verificate. Pe de-o parte codul metodei *medie* e partiţionat clar: la o execuţie normală se execută ce e în *try*; dacă apare excepţia *ExceptieNuMaiExistaNumere* se execută instrucţiunile din secţiunea *catch* corespunzătoare. Pe de altă parte, există şi un alt avantaj mai important: compilatorul ne obligă să spunem clar ce se întâmplă dacă apare vreo excepţie într-o metodă: fie o tratăm în acea metodă fie anunţăm apelantul prin clauza *throws* că metoda noastră se poate termina cu excepţii.

8.2.4 Emiterea explicită a excepţiilor

Să vedem acum cum anume se anunţă explicit apariţia unei situaţii neobişnuite. Mai exact, vom vedea cum anume se emite explicit o excepţie. Pentru acest lucru se utilizează instrucţiunea *throw*. Forma sa generală este prezentată mai jos, unde *ExpresieDeTipExceptie* trebuie să producă o referinţă la un obiect excepţie.

```
throw ExpresieDeTipExceptie;
```

Să vedem acum cum rescriem clasa *SirNumereReale* astfel încât ea să emită excepţii la întâlnirea situaţiilor neobişnuite.

```
class SirNumereReale {

    private double[] sir = new double[100];
    private int nr = 0;

    public void adauga(double x) throws ExceptieSirPlin {
        if(nr == sir.length) {
            throw new ExceptieSirPlin(''Sirul este plin!'');
        }
        sir[nr] = x;
        nr++;
    }

    public double extrageDupa(double x)
                  throws ExceptieNumarAbsent, ExceptieNuExistaNumere {
        int i;
        for(i = 0; i < nr; i++) {
            if(sir[i] == x) {
                if(i + 1 < nr) {
                    double result = sir[i + 1];
                    for(int j = i + 1; j < nr - 1; j++) {
                        sir[j] = sir[j + 1];
                    }
                    nr--;
                    return result;
                } else {
                    throw new ExceptieNuExistaNumere();
                }
            }
        }
        throw new ExceptieNumarAbsent(x);
    }
}
```

8.2.5 Situaţii în care se pot emite implicit excepţii

Multe instrucţiuni Java pot emite excepţii într-o manieră implicită în sensul că nu se utilizează instrucţiunea *throw* pentru emiterea lor. Aceste excepţii sunt emise de maşina virtuală Java în momentul detecţiei unei situaţii anormale la execuţie.

Spre exemplu, să considerăm un tablou cu 10 intrări. În momentul în care vom încerca să accesăm un element folosind un index mai mare ca 9 (sau mai mic ca 0) maşina virtuală Java va emite o excepţie de tip *IndexOutOfBoundsException*. Este important de menţionat că toate excepţiile ce sunt emise de maşina virtuală se propagă şi se interceptează exact ca orice alt fel de excepţie definită de un programator. În exemplul de mai jos prezentăm un mod (oarecum abuziv) de iniţializare a tuturor intrărilor unui tablou de întregi cu valoarea 1.

```
class ExempluUnu {

    public static void main(String argv[]) {
        int[] tab = new int[10];
        int i = 0;
        try {
            while(true) {
                tab[i++] = 1;
            }
        } catch(IndexOutOfBoundsException e) {
            //Aici se ajunge in momentul in care incerc sa
            //accesez elementul 10 al tabloului
            System.out.println(''Am initializat tot tabloul'');
        }
    }
}
```

Există multe alte instrucţiuni care pot emite implicit excepţii. Să considerăm codul de mai jos. Metoda returnează rezultatul împărţirii primului parametru la al doilea. Toată lumea ştie că împărţirea la 0 este o eroare. Dacă la un apel al metodei, b ia valoarea 0 atunci operaţia de împărţire ve emite (de fapt maşina virtuală) o excepţie de tip *ArithmeticException*. Este interesant de observat că aplicând algoritmul de propagare al excepţiilor prezentat anterior, metoda *divide* se va putea termina cu o excepţie *ArithmeticException*. Întrebarea e de ce compilează acest cod din moment ce nu avem clauza *throws*? Răspunsul va fi dat în secţiunea următoare.

```
class ExempluDoi {
```

```
    public static double divide(int a, int b) {
        return a / b;
    }
}
```

8.2.6 Clasificarea excepţiilor

În Java excepţiile se clasifică în excepţii verificate şi excepţii neverificate. Dacă o metodă se poate termina cu o excepţie verificată, tipul excepţiei respective (sau un supertip de-al său) trebuie să apară în clauza *throws* a respectivei metode. Altfel, la compilarea codului respectivei metode se va semnala o eroare. Dacă o metodă se poate termina cu o excepţie neverificată, nu este obligatorie prezenţa tipului său (sau a unui supertip de-al său) în clauza *throws* a metodei iar codul său va fi totuşi compilabil.

Excepţiile *IndexOutOfBoundsException*, *ArithmeticException* şi alte excepţii ce pot fi emise implicit de maşina virtuală sunt excepţii neverificate. Programatorul poate şi el defini şi emite prin instrucţiunea *throw* excepţii neverificate. Pentru a modela o astfel de excepţie clasa sa trebuie să moştenească direct ori indirect clasa *RuntimeException*. Toate celelalte excepţii a căror clasă nu au ca superclase directe ori indirecte clasele *RuntimeException* sau *Error* sunt excepţii verificate.

8.3 Exerciţiu rezolvat

Calcul număr de excepţii create

Să se creeze o clasă *MyException* derivată din clasa *Exception* ce conţine:

- un constructor ce are ca parametru un şir de caractere ce va fi furnizat de serviciul *getMessage()* al excepţiei. Serviciul *getMessage()* nu va fi suprascris.
- o metodă ce returnează de câte ori a fost instanţiată, atât ea (clasa *MyException*) cât şi orice subclasă a sa. După ce aţi implementat metoda precum şi mecanismul de numărare, explicaţi datorită cărui fapt metoda returnează şi câte instanţe ale subclaselor au fost create.

Creaţi într-o metodă *main* trei obiecte de tip *MyException* care vor fi ataşate pe rând aceleiaşi referinţe. Apelaţi pentru fiecare obiect creat două servicii furnizate de acesta.

Rezolvare

```
class MyException extends Exception {

    private static int instanceNo = 0;
```

```
    //acest constructor se apeleaza la fiecare instantiere a clasei,
    //precum si a eventualelor subclase
    public MyException(String message) {
        super(message);
        instanceNo++;
    }

    public static int getInstanceNo() {
        return instanceNo;
    }

    public static void main(String[] argv) {
        MyException e;
        //serviciile ce se doresc a fi apelate sunt
        //mostenite de la clasa Exception
        //getMessage(), toString(), printStackTrace(), ...

        //se poate apela si getInstanceNo(), dar aceasta fiind statica
        //nu se recomanda apelul prin intermediul referintei unui obiect

        e = new MyException("primul caz");
        System.out.println(e.getMessage());
        System.out.println(e.getInstanceNo());

        e = new MyException("al doilea caz");
        System.out.println(e.getMessage());
        System.out.println(e.toString());

        e = new MyException("al treilea caz");
        e.printStackTrace();
        System.out.println(e.toString());
    }
}
```

8.4 Exerciţii

1. Ce se va afişa pe ecran la execuţia programului de mai jos? Explicaţi de ce.

```
class L1 extends Exception {
    public String toString() {
        return "L1";
    }
}
```

```
class L2 extends Exception {
    public String toString() {
        return "L2";
    }
}

class Test {

    public static void main(String argv[]) {
        try {
            int i;
            for(i = 0; i < 4; i++) {
                if(i == 0) throw new L1();
                else throw new L2();
            }
        } catch(L1 e) {
            System.out.println(e);
        } catch(L2 e) {
            System.out.println(e);
        }
    }
}
```

2. Se dă codul de mai jos. Definiţi excepţiile verificate *E1* şi *E2* astfel încât codul clasei *Exemplu* să fie compilabil fără a-i aduce niciun fel de modificări.

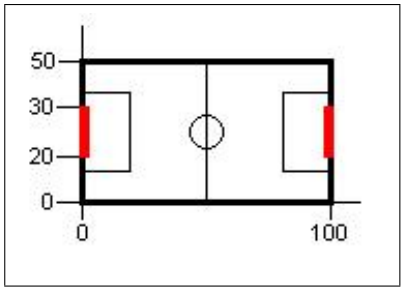
```
class Exemplu {

    public void doSomething(int i) {
        try {
            if(i == 0) throw new E1();
            else throw new E2();
        } catch(E1 e) {
         System.out.println("Prins");
        }
    }

}
```

3. Clasele *E1* şi *E2* sunt excepţii. Definiţi aceste excepţii în aşa fel încât codul de mai jos să fie compilabil fără a-i aduce niciun fel de modificare.

```
class Exemplu {
```

```
    public void doSomething(int i) {
        if(i == 0) throw new E1();
        else throw new E2();
    }
}
```

4. Să se scrie un program Java care modelează un meci de fotbal simplificat. Meciul se desfăşoară pe terenul de fotbal din figura de mai jos, teren ale cărui dimensiuni sunt indicate în aceeaşi figură.



Programul conţine o clasă *Minge* care are două atribute reprezentând poziţia (X,Y) curentă a mingii. Aceste coordonate se setează prin parametrii constructorului acestei clase. Clasa mai conţine două metode care permit determinarea coordonatei X respectiv Y a mingii, şi o metodă *suteaza()*. Această metodă generează două noi valori pentru poziţia mingii şi, în anumite situaţii, îşi va termina execuţia cu excepţii verificate. Astfel:

- dacă mingea ajunge într-o poziţie (X,Y) cu proprietatea Y=0 sau Y=50, se va genera o excepţie de tip *Out*.
- dacă mingea ajunge într-o poziţie (X,Y) cu proprietatea X=0 sau X=100 şi e gol (adică Y>=20 şi Y<=30) se va genera o excepţie de tip *Gol*.
- dacă mingea ajunge într-o poziţie (X,Y) cu proprietatea X=0 sau X=100, dar nu e gol sau out (adică 0<Y<20 sau 30<Y<50), atunci se va genera o excepţie de tip *Corner*.

Programul mai conţine o clasă *Joc* care va avea atribute pentru numele echipelor (setate la crearea unui obiect *Joc*), pentru numărul de goluri corespunzător fiecărei echipe şi pentru numărul total de out-uri şi cornere pe întregul meci. Clasa mai defineşte o metodă ce întoarce reprezentarea sub formă de şir de caractere a unui obiect *Joc*, reprezentare ce include numele echipelor, scorul şi statisticile descrise anterior. Desfăşurarea propriu-zisă a jocului se realizează de metoda *simuleaza()* din cadrul aceleiaşi clase.

Simularea constă în crearea unei mingi iniţiale urmată de efectuarea unui număr de 1000 de şuturi. Pentru fiecare poziţie ocupată de minge în timpul simulării se va afişa un mesaj de forma "Nume echipa 1 - Nume echipa 2 : Mingea se află la coordonatele (X,Y)". Tratarea situaţiilor excepţionale ce pot să apară constă în modificarea corespunzătoare a scorului şi a statisticilor, afişarea unui mesaj corespunzător pe ecran, precum şi de înlocuirea mingii curente cu una nouă, plasată după caz astfel:

- în caz de gol, se va creea o nouă minge amplasată la mijlocul terenului (X=50 şi Y=25).
- în caz de out, se va creea o nouă minge plasată la aceeaşi poziţie ca vechea minge.
- în caz de corner, noua minge se va plasa în colţul corespunzător al terenului.

Pentru exeplificare, într-o metodă main se vor creea două obiecte *Joc*, se vor simula ambele jocuri şi, în final, se vor afişa pe ecran rezultatele şi statisticile jocurilor.

NOTĂ
Pentru a obţine rezultate interesante se pune la dispoziţie clasa de mai jos care generează perechi (X,Y) reprezentând puncte de pe teren sau de pe frontierele acestuia.

```
import java.util.Random;
import java.util.Date;

class CoordinateGenerator {

    private Random randomGenerator;

    public CoordinateGenerator() {
        Date now = new Date();
        long sec = now.getTime();
        randomGenerator = new Random(sec);
    }

    public int generateX() {
        int x = randomGenerator.nextInt(101);
        if(x < 5) {
            x = 0;
        } else if(x > 95) {
            x = 100;
```

```
        } else {
            x = randomGenerator.nextInt(99) + 1;
        }
        return x;
    }

    public int generateY() {
        int y = randomGenerator.nextInt(101);
        if(y < 5) {
            y = 0;
        } else if(y > 95) {
            y = 50;
        } else {
            y = randomGenerator.nextInt(49) + 1;
        }
        return y;
    }
}
```

Bibliografie

1. James Gosling, Bill Joy, Guy L. Steele Jr., Gilad Bracha, *Java Language Specification*, *http://java.sun.com/docs/books/jls*, 2005.
2. Carmen De Sabata, Ciprian Chirilă, Călin Jebelean, *Laboratorul de Programare Orientată pe Obiecte*, Lucrarea 8 - Tratarea excepţiilor - Aplicaţii, UPT 2002, variantă electronică.