

## Lecția 8

# Tratarea excepțiilor

Un program comercial, fie el scris în Java sau în orice alt limbaj de programare, trebuie să țină cont de posibilitatea apariției la execuție a unor anumite situații neobișnuite: fișierul din care se dorește a se citi o informație nu există, utilizatorul a introdus un șir de caractere de la tastatură dar care nu reprezintă un număr așa cum a cerut programul, ș.a.m.d. În această lucrare vom studia un mecanism dedicat tratării acestor situații excepționale.

### 8.1 Excepțiile și tratarea lor

#### 8.1.1 Ce este o excepție?

Ideal, toate situațiile neobișnuite ce pot să apară pe parcursul execuției unui program trebuie detectate și 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 introducă un șir de caractere ce nu reprezintă un număr. Un program bine gândit va detecta această problemă, va anunța greșeala și probabil va cere utilizatorului reintroducerea numărului. Un program care se oprește subit în momentul în care utilizatorul face o greșeală, fără nici un avertisment și fără nici un mesaj clar care să anunțe problema, va “băga în ceață” orice utilizator.

Exemplul de mai sus este doar un caz particular de situație neobișnuită sau de excepție. Situațiile neobișnuite pot apare pe parcursul execuției programului și 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  $i$  și tabloul fiind specificați prin parametrii metodei. Dacă valoarea indexului depășește limitele tabloului, vorbim tot de o situație neobișnuită, deși ea are un iz de eroare de programare. Și aceste situații ar trebui detectate de un program chiar dacă utilizatorul nu prea are ce face în cazul unei erori de programare. Tratarea unei astfel de situații 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.

**Atenție**

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

Din exemplele de până acum s-ar putea crede că tratarea unei excepții implică într-o formă sau alta intervenția utilizatorului. Nu este deloc așa. O practică în industria aero-spațială (e drept destul de primitivă) este ca o aceeași parte dintr-un program să fie scrisă de două ori de programatori diferiți. Dacă la execuție programul de control al sistemului de zbor detectează o eroare de programare într-un exemplar al respectivei părți, el va cere automat celui de-al doilea exemplar să preia responsabilitățile primului exemplar. Acesta este doar un exemplu în care chiar și unele erori de programare pot fi tratate de un program. Să nu mai vorbim de situații mai simple. De exemplu, pentru a îndeplini anumite funcționalități un program trebuie să acceseze serviciile puse la dispoziție de un server. Se poate întâmpla ca la un moment dat respectivul server să fie suprasolicitat și să nu poată răspunde la cererea programului. Și o astfel de situație neobișnuită poate fi văzută ca o excepție la apariția căreia programul ar putea încerca, de exemplu, să acceseze automat un alt server care pune la dispoziție aceleași servicii.

În urma acestei discuții putem defini o excepție în felul următor:

**Definiție 1** *O excepție reprezintă un eveniment deosebit ce poate apare pe parcursul execuției unui program și care necesită o deviere de la cursul normal de execuție al programului.*

### 8.1.2 Metode clasice de tratare a excepțiilor

Pentru a ilustra modul clasic de tratare a situațiilor neobișnuite și a deficiențelor sale vom considera un exemplu ipotetic. Să presupunem un program care manipulează șiruri de numere reale. Clasa de mai jos modelează un astfel de șir. Pentru simplitate vom considera că asupra unui șir putem efectua două operații: adăugarea unui număr la sfârșitul său, respectiv extragerea din șir al primului număr de după prima apariție în șir a unui număr dat. O cerință suplimentară asupra acestei clase este ca diferite situații anormale să poată fi puse în evidență de apelantii 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 situațiile neobișnuite sunt tratate utilizând convenții. Astfel, apelantul metodei de adăugare își va putea da seama că operația a eșuat 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 situație neobișnuită atunci, pe baza valorii returnate de metodă, apelantul metodei își poate da seama ce s-a întâmplat: valoarea -1 spune apelantului că în acel șir 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 șir. Distincția între valoarea -1 ca număr din șir și valoarea -1 ca și cod de identificare a problemei apărute este realizată prin intermediul fanionului.

La prima vedere, aceste convenții par bune. Din păcate, ele fac utilizarea unui obiect șir destul de dificilă. Să considerăm acum că trebuie să scriem o metodă care primește ca parametru un număr și un șir și trebuie să extragă primele 10 numere ce apar în șir după prima apariție a numărului dat ca parametru și 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 convenție de codificare a unei situații neobișnuite ci doar o cerință 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 așa și codul totuși e compilabil! Ce se întâmplă dacă parametrul  $x$  nu există în șirul 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 distanță. 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 șir după parametrul  $x$ . Are dreptate. Programatorul ce a implementat metoda *medie* dă vina pe specificațiile primite care nu spuneau nimic de faptul că parametrul  $x$  ar putea să nu apară în șir. Are dreptate. Managerul de proiect dă vina însă tot pe el pentru că nu a sesizat această inconsistență și nu a anunțat-o. Programatorul replică prin faptul că a considerat că metoda se apelează totdeauna cu un șir ce-l conține 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 excepțiilor ridică o serie de probleme. Pe de-o parte codul poate deveni destul de greu de urmărit datorită nenumăratelor teste care trebuie realizate și care se întretes în codul esențial. Cea mai mare parte din codul metodei *medie* testează valoarea fanioanelor pentru a detecta situații neobișnuite și pentru a le trata corespunzător. Aceste teste se întrepătrund cu ceea ce face metoda de fapt: ia 10 numere din șir ce apar după parametrul *x* și le calculează media. Efectul este că metoda va fi foarte greu de înțeles pentru o persoană care vede codul prima dată. Pe de altă parte, cea mai mare problemă a modului clasic de tratare a excepțiilor este că el se bazează pe convenții: după fiecare apel al metodei *extrageDupa* trebuie verificat fanionul pentru a vedea dacă nu a apărut vreo situație neobișnuită. Din păcate convențiile 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 excepțiilor verificate pentru a anunța situațiile neobișnuite?

## 8.2 Tratarea excepțiilor în Java

### 8.2.1 Definirea excepțiilor

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 excepție nu este altceva decât un obiect care se definește aproape la fel ca orice alt obiect. Singura condiție suplimentară ce trebuie satisfăcută este să fie moștenită direct ori indirect clasa predefinită *Throwable* de către clasa obiectului nostru excepție. În practică, la definirea excepțiilor NU se extinde însă această clasă, ci se utilizează clasa *Exception*, o subclasă a clasei *Throwable*. Să vedem acum cum definim excepțiile pentru situațiile speciale din exemplul de la începutul lecției.

```
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 lecție am spus că obiectele ce compun un sistem software interacționează 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 situații: metoda s-a executat în

mod obișnuit sau metoda s-a terminat în mod neobișnuit datorită unei excepții. Clauza *throws* apare în antetul unei metode și ne spune ce tipuri de excepții pot conduce la **terminarea** neobișnuită a respectivei metode. Mai jos prezentăm modul în care specificăm faptul că metodele clasei *SirNumereReale* pot să se termine datorită unor situații neobișnuite și care sunt acele situații.

```
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 obișnuit o valoare *double* reprezentând un număr din sir. Clauza *throws* spune că în situații excepționale, metoda poate returna o referință la un obiect *ExceptieNumarAbsent*, o referință la un obiect *ExceptieNuExistaNumere* sau o referință la un obiect a cărui clasă (tip) moștenește clasele (tipurile) *ExceptieNumarAbsent* sau *ExceptieNuExistaNumere*.

**Atenție**

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

### 8.2.3 Interceptarea excepțiilor

Până acum am vorbit despre modul de definire a unei excepții și de modul de specificare a faptului că o metodă poate să se termine cu excepție. Dar cum poate afla un obiect ce apelează o metodă ce se poate termina cu excepție 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
} catch(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, secțiunile blocului *try-catch-finally* au următoarea utilizare. În secțiunea *try* se introduce codul pe parcursul căruia pot apare excepții. În fiecare secțiune *catch* se amplasează secvența de instrucțiuni ce trebuie să se execute în momentul în care în secțiunea *try* a apărut o excepție de tipul parametrului secțiunii *catch* sau de un subtip de-al său. În secțiunea *finally* se amplasează cod ce trebuie neapărat să se execute înaintea părăsirii blocului *try-catch-finally*, indiferent de faptul că în secțiunea *try* a apărut sau nu vreo excepție și indiferent dacă ea a fost interceptată ori nu de vreo secțiune *catch*.

Cum funcționează la execuție un bloc *try-catch-finally*? În situația în care nu apare nici o excepție în secțiunea *try* lucrurile sunt simple: se execută secțiunea *try* ca orice alt bloc de instrucțiuni, apoi se execută secțiunea *finally* dacă ea există și apoi se continuă cu prima instrucțiune de după blocul *try-catch-finally*. Prin urmare secțiunile *catch* NU se execută dacă nu apare vreo excepție în secțiunea *try*.

În continuare, pentru a înțelege cum funcționează un bloc *try-catch-finally* atunci când apar excepții, trebuie să înțelegem ce se întâmplă la execuție în momentul apariției unei excepții. În primul rând trebuie să spunem că la execuția programului, dacă se emite o excepție, execuția “normală” a programului e întreruptă până în momentul în care excepția respectivă este tratată (interceptată). Prinderea unei excepții înseamnă executarea unei secțiuni *catch* dintr-un bloc *try-catch-finally*, secțiune *catch* asociată acelei excepții. Din momentul emiterii și până în momentul prinderii se aplică în mod repetat un algoritm pe care-l vom prezenta în continuare. Algoritmul se aplică mai





În primul pas al algoritmului se verifică dacă respectiva instrucțiune/apel apare într-o secțiune *try* a unui bloc *try-catch-finally*. Dacă nu, respectiva metodă se termină automat cu aceeași excepție deoarece ea nu interceptează respectiva excepție. 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 secțiunea *try* a unui bloc înfășurător). Odată găsit, algoritmul baleiază în ordine textuală fiecare secțiune *catch* și verifică dacă tipul concret al excepției apărute este de tipul parametrului sau de un subtip de-al tipului parametrului. La găsirea primei clauze *catch* ce respectă această regulă căutarea se oprește și se trece la pasul trei. Dacă nu se găsește o astfel de secțiune *catch* se trece automat la execuția secțiunii *finally* (dacă ea există) după care se repetă pasul doi pentru un eventual bloc *try-catch-finally* a cărui secțiune *try* include blocul *try-catch-finally* curent. Dacă nu există un astfel de bloc metoda noastră se termină cu aceeași excepție ce a declanșat algoritmul deoarece ea nu o interceptează.

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



Dacă ați citit cu atenție veți constata că algoritmul nu spune nimic despre ce se întâmplă dacă se emite o excepție într-o secțiune *catch* sau *finally*. Lucrurile sunt simple: se aplică algoritmul ca pentru orice emisie de excepție. Situația mai puțin clară este atunci când se execută o secțiune *finally* și excepția care a declanșat algoritmul nu a fost încă tratată. Într-o astfel de situație, excepția curentă se consideră tratată și se reîncepe algoritmul pentru excepția nou emisă. Atenție însă la faptul că aceste excepții nu apar în secțiunea *try* a blocului *try-catch-finally* curent!!! Adică o excepție emisă într-o secțiune *catch* sau *finally* a unui bloc *try-catch-finally* nu poate fi prinsă de o secțiune *catch* a aceluiași bloc *try-catch-finally*!



Atenție 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 așa. În a doua clauză *catch* nu se ajunge niciodată! Deși în secțiunea *try* apare o excepție de tip *oExcepție*, ea va fi prinsă totdeauna de prima secțiune *catch* deoarece *oExcepție* e subtip de-al lui *Exception*. Din fericire, compilatorul sesizează astfel de probleme.

```
class oExcepție 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 lecției când folosim excepții.

```
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 execuția acestei metode. Presupunem inițial că numărul  $x$  există în șir și există 10 numere după el. Prin urmare metoda *extrageDupa* nu se va termina cu excepție, secțiunea *try* se va executa ca orice alt bloc de instrucțiuni iar după terminarea sa se continuă cu prima instrucțiune de după blocul *try-catch-finally*; adică se va executa instrucțiunea *return* și metoda noastră se termină normal.

Să vedem acum ce se întâmplă dacă după numărul  $x$  mai există în șir doar un număr. Prin urmare, la al doilea apel al lui *extrageDupa* metoda se va termina cu o excepție de tip *ExceptieNuExistaNumere*. Aplicând algoritmul prezentat anterior se constată că excepția a apărut într-o secțiune *try*. Se baleiază apoi secțiunile *catch* și se constată că prima este destinată prinderii excepțiilor de tipul celei emise. Prin urmare se trece la execuția corpului ei și *medie* va deveni 0. Apoi se trece la execuția primei instrucțiuni de după *try-catch-finally* și se execută instrucțiunea *return*. Este important de observat că în momentul în care *extrageDupa* s-a terminat cu excepție NU SE MAI EXECUTĂ NIMIC DIN SECȚIUNEA TRY. Pur și simplu se abandonează secțiunea *try* indiferent ce instrucțiuni sunt acolo. În cazul nostru se abandonează inclusiv ciclul *for* care încă nu s-a terminat când a apărut excepția.

Să vedem acum ce se întâmplă dacă nu avem numărul  $x$  în șir. Evident, *extrageDupa* se termină cu o excepție *ExceptieNumarAbsent* la primul apel. Se aplică algoritmul prezentat mai sus și se constată că această excepție nu e tratată de blocul *try-catch-finally* înconjurător deoarece nu există o clauză *catch* care să prindă această excepție. Prin urmare, metoda *medie* se va termina cu aceeași excepție. Acesta e și motivul pentru care apare tipul acestei excepții 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 secțiune *catch* corespunzătoare). Prin urmare, compilatorul vede că metoda *medie* s-ar putea termina cu această excepție și îi spune programatorului să se hotărască: fie interceptează excepția în această metodă fie specifică explicit că metoda se poate termina cu *ExceptieNumarAbsent*.



Uitați-vă la codul de mai jos. Codul nu e compilabil! Motivul e simplu. Clauza *throws* spune compilatorului că metoda se poate termina cu o excepție de tipul *oExceptie* sau cu o excepție de un subtip de-al său. Nimic nu-i poate schimba această părere. Pe de altă parte în metoda *altaMetoda* se prind doar excepții de tip *altaExceptie*. Acest lucru e corect: pot ajunge astfel de excepții aici. Altceva e problematic pentru compilator: se prind doar anumite cazuri particulare de excepții *oExceptie*. Ce se întâmplă cu restul? Ca urmare, compilatorul va semnala o eroare ce va fi rezolvată de programator fie prin adăugarea unei secțiuni *catch* pentru excepții de tip *oExceptie* după secțiunea deja existentă (mai știți 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 excepțiilor verificate. Pe de-o parte codul metodei *medie* e partiționat clar: la o execuție normală se execută ce e în *try*; dacă apare excepția *ExceptieNuMaiExistaNumere* se execută instrucțiunile din secțiunea *catch* corespunzătoare. Pe de altă parte, există și un alt avantaj mai important: compilatorul ne obligă să spunem clar ce se întâmplă dacă apare vreo excepție într-o metodă: fie o tratăm în acea metodă fie anunțăm apelantul prin clauza *throws* că metoda noastră se poate termina cu excepții.

### 8.2.4 Emiterea explicită a excepțiilor

Să vedem acum cum anume se anunță explicit apariția unei situații neobișnuite. Mai exact, vom vedea cum anume se emite explicit o excepție. Pentru acest lucru se utilizează instrucțiunea *throw*. Forma sa generală este prezentată mai jos, unde *ExpresieDeTipExcepție* trebuie să producă o referință la un obiect excepție.

```
throw ExpresieDeTipExcepție;
```

Să vedem acum cum rescriem clasa *SirNumereReale* astfel încât ea să emită excepții la întâlnirea situațiilor neobișnuite.

```
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 Situații în care se pot emite implicit excepții

Multe instrucțiuni Java pot emite excepții într-o manieră implicită în sensul că nu se utilizează instrucțiunea *throw* pentru emiterea lor. Aceste excepții sunt emise de mașina virtuală Java în momentul detecției unei situații anormale la execuție.

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) mașina virtuală Java va emite o excepție de tip *IndexOutOfBoundsException*. Este important de menționat că toate excepțiile ce sunt emise de mașina virtuală se propagă și se interceptează exact ca orice alt fel de excepție definită de un programator. În exemplul de mai jos prezentăm un mod (oarecum abuziv) de inițializare 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 instrucțiuni care pot emite implicit excepții. Să considerăm codul de mai jos. Metoda returnează rezultatul împărțirii primului parametru la al doilea. Toată lumea știe că împărțirea la 0 este o eroare. Dacă la un apel al metodei, *b* ia valoarea 0 atunci operația de împărțire va emite (de fapt mașina virtuală) o excepție de tip *ArithmeticException*. Este interesant de observat că aplicând algoritmul de propagare al excepțiilor prezentat anterior, metoda *divide* se va putea termina cu o excepție *ArithmeticException*. Întrebarea e de ce compilează acest cod din moment ce nu avem clauza *throws*? Răspunsul va fi dat în secțiunea următoare.

```
class ExempluDoi {
```

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

### 8.2.6 Clasificarea excepțiilor

În Java excepțiile se clasifică în excepții verificate și excepții neverificate. Dacă o metodă se poate termina cu o excepție verificată, tipul excepției 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 excepție neverificată, nu este obligatorie prezența tipului său (sau a unui supertip de-al său) în clauza *throws* a metodei iar codul său va fi totuși compilabil.

Excepțiile *IndexOutOfBoundsException*, *ArithmeticException* și alte excepții ce pot fi emise implicit de mașina virtuală sunt excepții neverificate. Programatorul poate și el defini și emite prin instrucțiunea *throw* excepții neverificate. Pentru a modela o astfel de excepție clasa sa trebuie să moștenească direct ori indirect clasa *RuntimeException*. Toate celelalte excepții a căror clasă nu au ca superclase directe ori indirecte clasele *RuntimeException* sau *Error* sunt excepții verificate.

## 8.3 Exercițiu rezolvat

### Calcul număr de excepții create

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

- un constructor ce are ca parametru un șir de caractere ce va fi furnizat de serviciul *getMessage()* al excepției. Serviciul *getMessage()* nu va fi suprascris.
- o metodă ce returnează de câte ori a fost instanțiată, atât ea (clasa *MyException*) cât și orice subclasă a sa. După ce ați implementat metoda precum și mecanismul de numărare, explicați datorită cărui fapt metoda returnează și câte instanțe ale subclaselor au fost create.

Creați într-o metodă *main* trei obiecte de tip *MyException* care vor fi atașate pe rând aceleiași referințe. Apelați 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 Exerciții

1. Ce se va afișa pe ecran la execuția programului de mai jos? Explicați 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. Definiți excepțiile verificate *E1* și *E2* astfel încât codul clasei *Exemplu* să fie compilabil fără a-i aduce niciun fel de modificări.

```
class Exemplu {

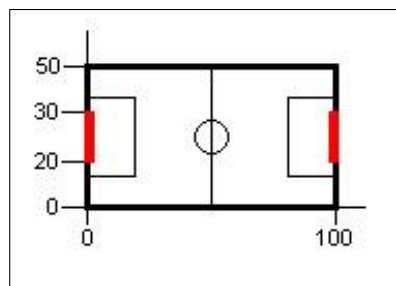
    public void doSomething(int i) {
        try {
            if(i == 0) throw new E1();
            else throw new E2();
        } catch(E1 e) {
            System.out.println("Prins");
        }
    }
}
```

3. Clasele *E1* și *E2* sunt excepții. Definiți aceste excepții în așa 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 desfășoară pe terenul de fotbal din figura de mai jos, teren ale cărui dimensiuni sunt indicate în aceeași figură.



Programul conține o clasă *Minge* care are două atribute reprezentând poziția (X,Y) curentă a mingii. Aceste coordonate se setează prin parametrii constructorului acestei clase. Clasa mai conține două metode care permit determinarea coordonatei X respectiv Y a mingii, și o metodă *suteaza()*. Această metodă generează două noi valori pentru poziția mingii și, în anumite situații, își va termina execuția cu excepții verificate. Astfel:

- dacă mingea ajunge într-o poziție (X,Y) cu proprietatea  $Y=0$  sau  $Y=50$ , se va genera o excepție de tip *Out*.
- dacă mingea ajunge într-o poziție (X,Y) cu proprietatea  $X=0$  sau  $X=100$  și e gol (adică  $Y \geq 20$  și  $Y \leq 30$ ) se va genera o excepție de tip *Gol*.
- dacă mingea ajunge într-o poziție (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 excepție de tip *Corner*.

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

Simularea constă în crearea unei mingi inițiale urmată de efectuarea unui număr de 1000 de șuturi. Pentru fiecare poziție ocupată de minge în timpul simulării se va afișa un mesaj de forma “Nume echipa 1 - Nume echipa 2 : Mingea se află la coordonatele (X,Y)”. Tratarea situațiilor excepționale ce pot să apară constă în modificarea corespunzătoare a scorului și a statisticilor, afișarea unui mesaj corespunzător pe ecran, precum și de înlocuirea mingii curente cu una nouă, plasată după caz astfel:

- în caz de gol, se va crea o nouă minge amplasată la mijlocul terenului (X=50 și Y=25).
- în caz de out, se va crea o nouă minge plasată la aceeași poziție ca vechea minge.
- în caz de corner, noua minge se va plasa în colțul corespunzător al terenului.

Pentru exeplicare, într-o metodă main se vor crea două obiecte *Joc*, se vor simula ambele jocuri și, în final, se vor afișa pe ecran rezultatele și statisticile jocurilor.

### NOTĂ

Pentru a obține rezultate interesante se pune la dispoziție 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 excepțiilor - Aplicații, UPT 2002, variantă electronică.