

Programozási nyelvek II.: JAVA

5. gyakorlat

2017. október 9-13.

- Kivételkezelés alapjai
- Be- és kimenet
- BufferedReader, Scanner
- PrintWriter

Mi is az a kivételkezelés?

- „Kivételes események” kezelése, mikor a program normális utasításfolyama megszakad.
- Amikor az esemény bekövetkezik egy metódusban, az eljárás létrehoz, „dob” egy különleges kivétel objektumot, és átadja azt a futtató környezetnek, a Java virtuális gépnek.
- A virtuális gép ezután megpróbál a hívási lánc mentén találni valamilyen kivételkezelési eljárást, melyet meghívhat.
- Amennyiben nincs ilyen metódus, és a hívási lánc mentén eljutunk a *main* metódusig, ahol szintén nem kerül lekezelésre a kivétel, akkor a futás megszakad.

```
// Main.java
public class Main {
    public static void main(String[] args) {
        int i = Integer.parseInt(args[0]);
        print(args[1], i);
    }

    public static void print(String text, int times) {
        Repeater repeater = new Repeater();

        // repeater.text = text;

        repeater.println(times);
    }
}
```

```
// Repeater.java
public class Repeater {
    public String text = null;

    public void println(int times) {
        text = text.trim();
        // white space levágása a szöveg elejéről és végéről

        for (int i = 0; i < times; ++i) {
            System.out.println(text);
        }
    }
}
```

```
$ javac Main.java
```

```
$ java Main 10 hello
```

```
Exception in thread "main" java.lang.NullPointerException  
    at Repeater.println(Repeater.java:6)  
    at Main.print(Main.java:13)  
    at Main.main(Main.java:5)
```

- Mivel a *repeater* objektum *text* mezőjét *null* értékűnek hagytuk, az objektumszintű *trim* metódus hívása *NullPointerException*-t váltott ki, hiszen a *null* referenciához nem tartozik objektum, nem tudjuk min meghívni a metódust.
- Mivel a kivételt nem kezeltük le, az egészen a *main* metódusig jutott, onnét pedig továbbszivároghva leállította a program futását.
- Ilyenkor a virtuális gép információt ad a kivételről, mint láttuk.

Hívási lánc (stack trace)

```
Exception in thread "main" java.lang.NullPointerException
    at Repeater.println(Repeater.java:6)
    at Main.print(Main.java:13)
    at Main.main(Main.java:5)
```

- A legfelső sor azt a metódust és azon belül azt a sort adja meg a kódban, ahol a kivétel keletkezett.
- A további sorok azt, hogy az eggyel fentebb lévő metódust honnan hívtuk, mikor a kivétel keletkezett.
- A változók értékéről ebből nem kapunk információt, ahhoz más megoldást kell keresnünk, például *debugger*t vagy logolást.

Figyelem!

- A kivételkezelés nem azonos a hibakezeléssel. Bár leggyakrabban erre a célra használjuk, egy kivételes esemény nem feltétlenül hiba.
- Például ha meg akarunk nyitni egy fájlt, mely nem létezik a fájlrendszeren, kivételt kapunk, viszont ez a konkrét feladattól függ, hogy hibának minősül-e. Ha mondjuk a fájl jelentené a programunk bemenetét, akkor a fájl hiánya valóban hiba. Ha viszont valamilyen konfigurációs fájlt tölténénk be a program futása elején, a fájl hiánya esetén létrehozhatjuk az alapértelmezett beállításokkal.
- A hibakezelésre Java-ban legtöbbször kivételeket használunk.

- Nem a fő végrehajtási ág.
- Logikailag alacsonyabb rendű, nem az adott problémát megoldó algoritmushoz kapcsolódó hiba, vagy a feladat megoldásához közvetlen nem kötődő, ritkán bekövetkező speciális esemény.
- pl.
 - nullával való osztás,
 - tömb túlindexelése,
 - hálózati kapcsolat hibája,
 - hiányzó fájl,
 - castolási hiba,
 - hibás számformátum, ha Stringet számmá alakítanánk.

```
try {  
    // ...  
} catch (ExceptionType1 e) {  
    // ...  
} catch (ExceptionType2 e) {  
    // ...  
} /*  
    ...  
*/ catch (ExceptionTypeN e) {  
    // ...  
} finally {  
    // ...  
}
```

- A *try* blokkba írjuk a védendő kódot, azt, amelyik kiválthatja a kivételt.
- Ezt egy vagy több *catch* ág követi, melyek kódja abban az esetben fut le, ha a megadott kivétel kiváltódik a *try* blokkban (vagy egy onnan meghívott metódusban, ahonnet továbbadásra kerül).
- Végül egy opcionális *finally* blokk következik, ezt a Java **minden körülmények között** lefuttatja, utolsóként. Tehát akkor is, ha
 - a *try* blokk lefutott, és nem keletkezett kivétel;
 - a *try* blokkban keletkezett egy kivétel, és az valamelyik *catch* lekezelte;
 - a *try* blokkban keletkezett egy kivétel, de azt nem tudtuk lekezelni, és továbbadjuk.

```
public class Main {  
  
    public static void main(String[] args) {  
        try {  
            System.out.println("Hello, " + args[0] + "!");  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.err.println(  
                "Not enough command line arguments.");  
            // standard error kimenetre írunk  
        } finally {  
            System.out.println("bye");  
        }  
    }  
}
```

```
$ javac Main.java
```

```
$ java Main
```

```
Not enough command line arguments.
```

```
bye
```

```
$ java Main World
```

```
Hello, World!
```

```
bye
```

- Input-output, avagy be- és kimenet.
- Nem csak fájlok beolvasását és kiírását értjük alatta, számos más módon is kaphat a programunk bemenetet és adhat kimenetet, például konzolon vagy hálózaton át.
- A Java egy több rétegű IO modellel rendelkezik, mely alkalmas a különböző források és eltérő célok egységes kezelésére.
- Ebből mi csak a fájlkezelést tekintjük most át.

- Olvassuk be egy parancssori argumentumként megadott fájl tartalmát, és írjuk ki a képernyőre!
- Olvassuk be a fájl tartalmát, és írjuk ki egy másik fájlba!

- Először szükségünk lesz egy *java.io.File* objektumra, mely egy (nem feltétlenül létező) fájlt (vagy könyvtárat) reprezentál a fájlrendszeren. Olyan, mint egy hivatkozás vagy link az adott fájlra, nem ellenőrzi, hogy a cél megvan-e. (Ne felejtsünk el importálni!)

```
String filepath = args[0]; // abszolút vagy relatív út
File file = new File(filepath);
```

- Létrehozunk egy *java.io.FileReader* objektumot, melynek konstruktora paraméterül kapja a *File* objektumunkat, és megnyitja a fájlt szerkesztésre.
- Mivel ez egy erőforrást (fájlt) kezelő típus, úgynevezett try-with-resource szerkezetbe ágyazva hozzuk létre, a *try* kulcsszó után gömbölyű zárójelek között. Így a *try* blokk elhagyásával az erőforrást (fájlt) a virtuális gép automatikusan elengedi.

```
try ( FileReader reader = new FileReader(file) ) {
    // itt használhatjuk a reader objektumot
}
```

- Mivel a *FileReader* csak alacsony szintű funkcionalitást biztosít, ezért ezt még becsomagoljuk egy *java.io.BufferedReader* objektumba, mellyel már tudunk majd soronként beolvasni.

```
try (FileReader reader = new FileReader(file);
    BufferedReader br = new BufferedReader(reader) ){

    // ...

}
```

- Vagy kicsit egyszerűbben:

```
try (BufferedReader br =
    new BufferedReader(new FileReader(file)) ){

    // ...

}
```

- A fenti kód fordításakor a fordító jelzi nekünk, hogy két kivételt mindenképpen le kell kezelnünk a fájl megnyitásakor. Ez a *java.io.FileNotFoundException* és a nála általánosabb *java.io.IOException*.

```
try (BufferedReader br =
    new BufferedReader(new FileReader(file))) {

    // ...
} catch (FileNotFoundException e) {
    System.err.println("File does not exist.");
} catch (IOException e) {
    System.err.println("An IO error occurred.");
}
```

- Mivel az *IOException* egy általánosabb kivétel, valójában elég lenne csak azt lekezelni. Ekkor *FileNotFoundException* esetén is az *IOException* kezelő *catch* ág futna le.

BufferedReader – readLine

- A fájlból való beolvasást a *BufferedReader* paraméter nélküli *readLine* metódusával tudjuk megtenni.
- Ha nincs több beolvasható sor, *nullt* ad vissza.

```
try ( BufferedReader br =
    new BufferedReader(new FileReader(file)) ) {

    String line;
    for (line = br.readLine();
        line != null;
        line = br.readLine()) {

        System.out.println(line);
    }
} catch (FileNotFoundException e) {
    System.err.println("File does not exist.");
} catch (IOException e) {
    System.err.println("An IO error occurred.");
}
```

- Egy másik megoldás a beolvasásra a *java.util.Scanner* osztály használata.
- Itt nincs szükség *FileReader* létrehozására, és az *IOException* sem kell tudnunk lekezelnünk, csak a *FileNotFoundException*. „Cserébe” viszont minden sor beolvasása előtt ellenőriznünk kell, hogy van-e még további sor.

```
try ( Scanner sc = new Scanner(file) ) {
    while (sc.hasNextLine()) {
        String line = sc.nextLine();
        System.out.println(line);
    }
} catch (FileNotFoundException e) {
    System.err.println("File does not exist.");
}
```

- Vegyük észre, hogy a beolvasó metódus neve is más! (*nextLine* az eddigi *readLine* helyett)

- A fájllokba való kiíratás elég hasonló, csak egy másik osztályra lesz szükségünk, ez a *PrintWriter*. A konstruktornak itt is egy *java.io.File* objektumot kell átadnunk. Itt természetesen nem kell, hogy a fájl létezzen, a Java mégis *FileNotFoundException* kivétellel jelzi, ha a fájl megnyitása vagy létrehozása sikertelen.
- A fájlba írni a *print* és *println* metódusokkal tudunk.
- A try-with-resource szerkezet használata és így a fájl lezárása itt még fontosabb, mivel anélkül a Java nem valójában írja ki a fájlba a sorokat, csak a memóriában tárolja őket.
- *Megjegyzés:* Alternatív megoldásként a *flush* metódussal írathatjuk ki a sorokat.

PrintWriter - 2.

```
String[] linesToWrite = // ...

try ( PrintWriter pw = new PrintWriter(file) ) {

    for (String line : linesToWrite) {
        pw.println(line);
    }
} catch (FileNotFoundException e) {
    System.err.println("File cannot be opened.");
}
```

Beolvasás fájlból, és kiíratás egy másikba (teljes kód)

```
import java.io.*;

public class FileCopy {

    public static void main(String[] args) {

        if (args.length < 2) {
            System.err.println(
                "Not enough command line arguments.");
        }

        String inFilename = args[0];
        String outFilename = args[1];

        File inFile = new File(inFilename);
        File outFile = new File(outFilename);

        ...
    }
}
```


Beolvasás fájlból, és kiíratás egy másikba (teljes kód) – 2.

```
...
    try ( BufferedReader br =
            new BufferedReader(new FileReader(inFile));
        PrintWriter pw = new PrintWriter(outFile) ) {

        String line;
        for (line = br.readLine();
            line != null;
            line = br.readLine() ) {

            pw.println(line);
        }
    } catch (FileNotFoundException e) {
        System.err.println("File cannot be opened.");
    } catch (IOException e) {
        System.err.println("An IO error occurred.");
    }
}
}
```

- Ha bizonyos kivételeket egységesen akarunk lekezelni, a metódusnál, melyből tovább akarjuk engedni őket, jeleznünk kell egy *throws* kulcsszó után.

```
import java.io.*;

public class FileCopy2 {

    public static void main(String[] args) {

        try {
            String contents = readFromFile(args[0]);
            printToFile(args[1], contents);
        } catch (IOException e) {
            System.err.println("An IO error occurred.");
        }
    }

    ...
}
```

Throws – 2.

...

```
private static String readFromFile(String filename)
    throws FileNotFoundException, IOException {

    try ( BufferedReader br = new BufferedReader(
        new FileReader(new File(filename))) ) {

        String line;
        StringBuilder builder = new StringBuilder();
        for (line = br.readLine(); line != null;
            line = br.readLine() ) {

            builder.append(line).append("\n");
        }
        return builder.toString();
    }
}
```

...

Throws – 3.

...

```
private static void printToFile(String filename,
    String contents) throws FileNotFoundException {

    try ( PrintWriter pw =
        new PrintWriter(new File(filename)) ) {

        pw.print(contents);
        pw.flush();
    }
}

}
```

Feladatok



- Elérhető ezen a **LINKEN** egy Java forrásfájl, mely a következő feladatot volna hivatott elvégezni. Javítsd ki a benne található hibákat!
- A főprogram parancssori argumentumként két fájlnevet kap. Az első fájl tartalmát átmásolja a másodikba, egy megszorítással.
- A másolás során ha kettő vagy több egymást követő sort talál, melyek karakterről karakterre megegyeznek, akkor ezek közül csak egyet ír ki.

- A bemeneti fájlunk sorai vesszővel elválasztott egész számokat tartalmaznak. Soronként adjuk össze őket, és írjuk ki egy másik fájlba!
- Oldjuk meg *BufferedReader*rel!
- Keressünk a *String* osztályban olyan metódust, mely alkalmas rá, hogy egy speciális karakter (most a vessző) mentén feldarabolja a sorunkat.
- Feltehetjük, hogy a bemenet formátuma helyes.

- Egy parancssori argumentumként megadott fájlban keressünk meg egy kapott szövegrészletet!
- A szövegrészletet kérjük be a felhasználótól a képernyőről.
- Írjuk ki, hogy hányszor fordult elő a keresett szövegrészlet a fájlban.
- Ne csak akkor számítsuk találatnak, ha az egész sossal megegyezik, akkor is vegyük figyelembe, ha a sor csak tartalmazza a keresett szövegrészletet! (Keressünk megfelelő metódust a *String* osztályban!)

- A bemeneti fájlunk sorai vesszővel elválasztott egész számokat tartalmaznak. Soronként adjuk össze őket, és írjuk ki egy másik fájlba!
- Oldjuk meg *Scanner*rel, oly módon, hogy rögtön az egész számokat olvassuk be, egyesével, nem pedig a sorokat daraboljuk fel.
- Ehhez nézzük meg a *Scanner* osztály *hasNextInt*, *nextInt* és *useDelimiter* metódusait.
- Feltehetjük, hogy a bemenet formátuma helyes.

- A bemeneti fájlunk sorai vesszővel elválasztott egész számokat tartalmaznak. Soronként adjuk össze őket, és írjuk ki egy másik fájlba!
- A fájl tartalmazhat hibás adatokat (nem számokat), ezeket a program egyszerűen hagyja figyelmen kívül!
- Oldjuk meg *BufferedReader*rel és *Scanner*rel is.