I/O eszközök
Kivételkezelés

Java PROGRAMOZÁS

2. GYAKORLAT

Bemenet és kimenet általában I.

- **Alapeset:** szeretnénk egy programhoz egy fájlt használni akár kimenet, akár bemenetként. Erre triviális megoldás lenne a szabványos kimenet/bement átirányítása. Azonban ha egy programban több fájlt akarunk használni, akkor a programon keresztül is kell tudnunk kimenetet, bemenetet kezelni.
- **A Java megoldása erre:** `java.io` csomag
- Ebben nem csak fájlkezelést találunk, hanem általános, **bemeneti és kimeneti adatfolyamok (streamek)** kezelését.
  - **Adatfolyam: valamilyen adatok sorozata**

Bemenet és kimenet általában II.

- A **bemeneti adatfolyam**hoz mindig egy adatforrás tartozik: adatokat lehet egymás után kiolvasni róla
  - Pl. : konzol, fájl, hálózati kapcsolat, adatbázis, bájt- ill. karaktertömb, stb…
- A **kimeneti adatfolyam**hoz mindig egy adatnyelő tartozik: adatokat lehet egymás után írni rá
  - Pl.: képernyő, fájl, hálózati kapcsolat, nyomtató, bájt- ill. karaktertömb, stb…
- Javában az adatfolyamoknak (streameknek) objektumok felelnek meg, és minden adatfolyam típusnak egy bizonyos osztály.

Az adatfolyamok (streamek)

- Az adatfolyam **irány**a alapján vannak:
  - bemeneti osztályok
  - kimeneti osztályok
- Az adatfolyamot alkotó adatok **típus**a alapján vannak:
  - bájtszervezésű adatfolyamok: 8 bit
  - karakterszervezésű adatfolyamok :16 bit (Unicode)
- **Feladat**uk szerint vannak:
  - forrás osztályok
  - nyelő osztályok
  - egy adatfolyamot további funkcionalitással ellátó osztályok (szűrők)

Elvárásaink

- Jogosan elvárhatjuk, **hogy ugyanazokat a műveleteket tudjuk elvégezni a különböző fajta be- és kimeneteken**.
- Mindenképpen előnyös, ha a különféle adatfolyam-objektumok (legyen az egy hálózati kapcsolat, vagy egy konzol) ugyanolyan, vagy legalábbis **hasonló kezelőfelülettel rendelkeznek**.
- Erre ad megoldást a jól strukturált `java.io` csomag, ami nagyfokú **rugalmasságot biztosít, és a streamek egyszerű kezelését hordozza magában.**

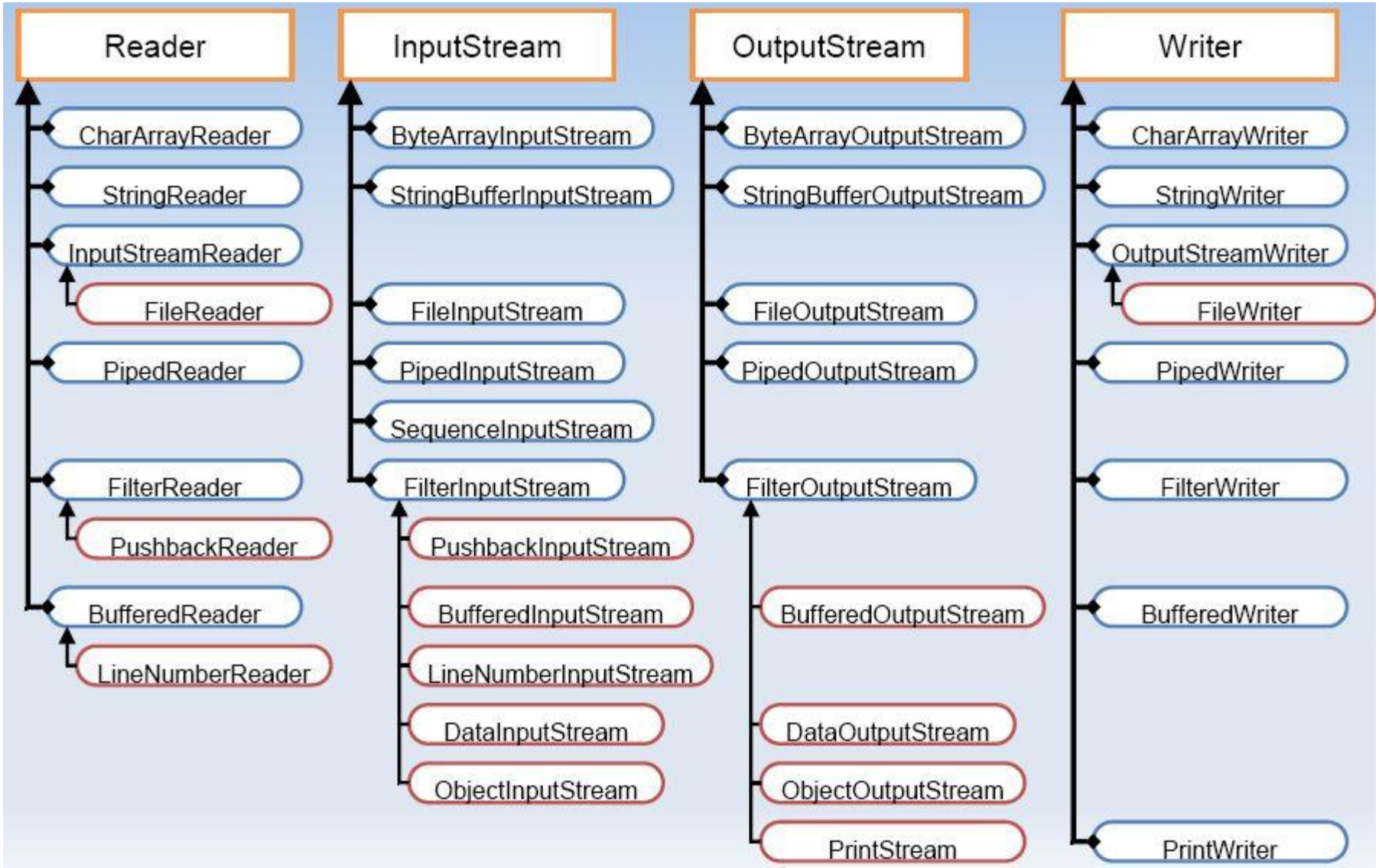
A `java.io` csomag bázis adatfolyam osztályai

- Az adatfolyam osztály hierarchia legfelső szintjén lévő 4 absztrakt bázisosztály:

Típus/Irány	Bemeneti	Kimeneti
Bájt	InputStream	OutputStream
Karakter	Reader	Writer

- A többi adatfolyam osztály ezekből leszármaztatva jött létre, az elnevezésük alapján ez szépen nyomon követhető.
  - Ezek pici osztályok, amelyek mind valamilyen kis funkcionalitást határoznak meg

A 4 bázisosztály és leszármazottaik:



Szűrők

- Az adatfolyam osztályok egyik csoportja a használt adatforrást, ill. adatnyelőt határozza meg (pl. `FileInputStream`, `FileReader`)
- Az adatfolyam osztályok másik csoportját az ún. **szűrők** alkotják, melyek segítségével egy meglévő adatfolyamot speciális tulajdonságokkal ruházhatunk fel.
  - Pl.: pufferelt beolvasás (`BufferedReader`), szöveges formátumú kiírás (`PrintWriter`)
- Egy szűrő adatfolyam objektumot mindig egy már meglévő adatfolyam-objektum köré kell létrehoznunk. Például:
- `FileWriter fw = new FileWriter("a.txt");`
- `PrintWriter pw = new PrintWriter(fw);`

System.out, System.in

- Már eddig is használtunk adatfolyam osztályokat: `System.out`, `System.in`. Ezek előre definiált változók
  - `System.out: PrintStream`, ez lényegében a standard kimenet.
  - `System.in: InputStream`, ez lényegében a standard bemenet.
- Van még: `System.err` is (hibajelzés).

Figyelem!

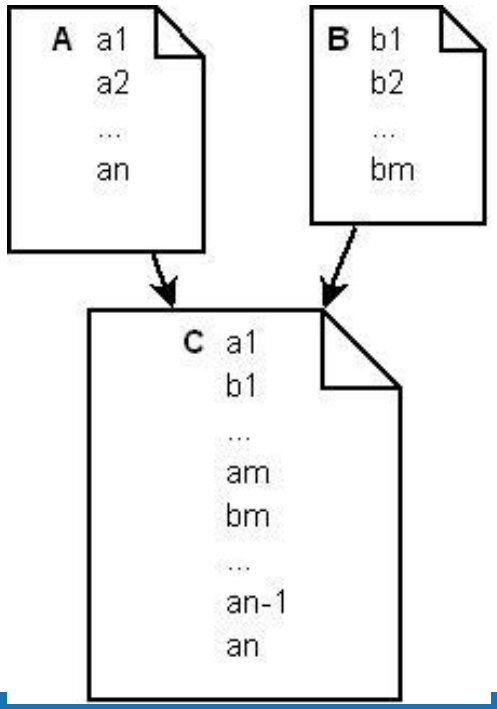
- `java.io.IOException`: **A** `java.io` **csomagba tartozó** csatornák kezelése a `java.io.IOException` **kivételt** válthatja ki. Kivételkezelésről később bővebben, addig is figyeljünk rá oda...
- **Adatfolyam lezárása: Fontos, hogy figyeljünk** arra, hogy adatfolyamainkat a használat után lezárjuk. Ez fontos, főleg a kimeneti adatfolyamok esetén, ugyanis a lezárással egyben gondoskodunk arról is, hogy amit az adatfolyamra akartunk írni, az tényleg oda is kerüljön (pufferek ürítése).
  - `close()` függvény.
  - A pufferek ürítését a `flush()` függvénnyel is elvégezhetjük.

Első feladat

- Mindenki importálja a 02_Java_IO_Exceptions projektet, és nyissa meg az io csomagot.
- Itt találtok egy *ReadWrite.java* nevű fájlt.
- Feladat a függvények megírása.
- A `main`-hez ne nyúljatok.

Gyakorló feladat

- Nyissuk meg a practice csomagban a Merge.java fájlt! Adottak az *a.txt* és a *b.txt* fáljok, fésüljük őket össze a *c.txt* fájlba úgy, hogy felváltva olvasunk belőlük!
- Figyeljünk arra, mi van akkor, ha az egyik fájlban már nincs több adat, de a másikban még van, illetve ha az egyik fájl esetleg üres!
- `br.ready();`
  - //leellenőrzi, hogy olvasható-e még a stream



Character

- `Character`: a `java.lang.Character` csomagoló osztály a `char` primitív típus köré, Unicode specifikáció szerint tartalmazza a karaktereket: http://www.unicode.org/Public/UNIDATA/UnicodeData.txt
- Tagfüggvényei:
  - `boolean isLetter(char c),`
  - `boolean isDigit(char c),`
  - `boolean isWhitespace(char c),`
  - `boolean isUpperCase(char c),`
  - `boolean isLowerCase(char c),`
  - `char toUpperCase(char c),`
  - `char toLowerCase(char c)`
- Escape sequence: amikor egy \ előz meg egy karaktert, ennek speciális jelentése van a fordítónak(‘\n’,’\t’,stb.).

Printf

- A `java.io.PrintStream.printf` függvény szövegek egyszerű formázását teszi lehetővé, egy format stringnek megfelelően.
- Szintaxisa:
  %[argument_index$][flags][width][.precision]conversion
- https://docs.oracle.com/javase/8/docs/api/java/io/PrintStream.html#printf-java.lang.String-java.lang.Object...-
  - `System.out.printf("Local time: %tT", Calendar.getInstance());`
    `=> Local time: 15:49:25`
  - `System.out.printf("%4$2s %3$2s %2$2s %1$2s", "a","b", "c", "d"); => "d c b a”`
- Nézzük meg a `printf` package `PrintfExample` osztályát!

Szövegkezelés - String

- Nincs primitív string típus, hanem a `String`, `StringBuilder` és `StringBuffer` osztályokat használhatjuk.
- A `String` Unicode karakterek sorozata.
  - `String s = "Szia!";`
- A `String` objektum tartalmát nem lehet módosítani (immutable), helyette új `String`-et kell létrehozni:
  - `s = "Helló!";`
- Hasznos tagfüggvények (mindig új objektum jön létre!):
  - `substring()`: `String` egy része kivágható
  - `split()`: a `String`-et egy reguláris kifejezéssel darabokra vágja, `String` tömbben kapjuk vissza a darabokat.
  - `trim()`: levághatók a whitespace-ek a `String` elejéről és végéről
  - `toLowerCase()`, `toUpperCase()`

equals vs ==

- Objektumok **egyenlőségét** az `equals()` metódussal **kell** vizsgálni.
- Az == azt mondja meg, hogy két referencia mikor egyenlő. (Azaz, hogy a két objektum mikor azonos.)
  - Pl.: nem lesz igaz az alábbi:
    - `new String("Hello world") == new String("Hello world")`
- Általában az osztályokban szokás felüldefiniálni az equals metódust, megadván, hogy mikor egyenlő két objektum. Ez például így már igaz:
  - `new String("Hello world").equals(new String("Hello world"))`
  - Ebben az esetben a `hashcode()` tagfüggvényt is ajánlatos felüldefiniálni.
- Operátortúlterhelés NINCS!
- FIGYELEM! ha a későbbiekben két `String` objektumot össze akarunk hasonlítani (hogy ugyan azon karakterek sorozatából állnak-e), akkor az `equals()` metódust használjuk!

Szövegkezelés – StringBuilder, StringBuffer

- A `StringBuilder` Unicode karakterek sorozata. A tartalmát meg lehet változtatni anélkül, hogy új objektumot kellene létrehozni. Más a reprezentációja a `String` osztályhoz képest, ezért más műveleteket lehet rajta hatékonyan megvalósítani.
- Ilyenek pl.:
  - `append(); insert(); reverse(); setCharAt(); setLength()`
- A műveletek, azon kívül, hogy transzformálják az objektumot, vissza is adnak egy referenciát rá:
  - ```
    StringBuffer sb = new StringBuffer("Szia!");
    sb.append(" Hi!").append(" Üdv!").reverse();
    ```
- A `StringBuffer` hasonló a `StringBuilder`-hez, de a metódusai szinkronizáltak (később...). Többszálú alkalmazásoknál ajánlott ezt használni, viszont lassabban működik, mint a `StringBuilder`.
- Futtassuk a `string` package-ben lévő *ConcatString1-2-3* java fájlokat! Mit tapasztalunk a sebesség terén?
    ```

Reguláris kifejezések

- Van, hogy egy inputot (pl.: `String`) szeretnénk egy logika szerint elfogadni, vagy tördelni. Ehhez egy döntési halmazt (reguláris kifejezést), szabályrendszert kell definiálni, amely szerint történik az elfogadás/tördelés. Ilyenek lehetnek pl.:
  - e-mail cím elfogadás
  - valamilyen karakterek szerinti tördelés
- Tehát mindenképpen meg kell adnunk egy halmazt, ami szerint „dönteni” kívánunk. A reguláris kifejezések segítségével komplex keresési mintákat adhatunk meg.
- Pl: keressünk olyan kifejezést, ami pontosan 2 karakterből áll.
  - regex: ”.. ”
- Pl: keressünk olyan kifejezést, ami csak számokból áll.
  - regex: ”\d+”

Reguláris kifejezések – Miket használhatunk?

- Egy karakter (ha nem speciális) saját magára illeszkedik.
- Speciális karaktereknél a \ karaktert kell használnunk. Pl. a * illeszkedik a * karakterre, \\ a \ karakterre.
  - FIGYELEM! A Java számára a \ karaktert külön escape-elni kell, ezért minden \ helyet \\-t kell írnunk!(Pl. \w→\\w, *→*, \\→\\\\)
- Karakter-osztályok (Character Classes) /vannak-e bizonyos karakterek a `String`-ünkben/:
  - . : bármilyen karakter szerepel
  - [abc]: a,b,c szerepel
  - [^abc]: a,b,c kivételével bármelyik szerepel
  - [a-zA-Z] : kis és nagybetű közül bármelyik szerepel
  - [a-dm-p]: a-tól d-ig VAGY m-től p-ig
  - [0-9]: szám (ugyan az \d)

Reguláris kifejezések – Miket használhatunk?

- Előredefiniált Karakter-osztályok (Predefined Character Classes):
  - . : bármilyen karakter szerepelhet
  - \d: digit [0-9]
  - \D: nem digit [^0-9]
  - \s: whitespace karakter [\t\n\x0B\f\r]
  - \S: nem whitespace karakter [^s]
  - \w: word karakter [a-zA-Z_0-9]
  - \W : nem word karakter [^w]
- Boundary Matchers (bizonyos karakterek hol vannak a stringben):
  - ^: sor eleje
  - $: sor vége
  - \b: word korlátosság pl.: \bdog\b és „there is a dog” versus „the doggie plays”
  - \A: input kezdete
  - \G: előző találat eredménye pl.: \Gdog és „dog dog”
  - \z: input vége

Reguláris kifejezések

- Reguláris kifejezéseket egymás után is írhatunk, ez a konkatenálást jelenti.
- Ha X és Y reguláris kifejezések, akkor X|Y azokra a sztringekre illeszkedik, melyek az X vagy Y mintára illeszkednek
- Az X+ kifejezés az X reguláris kifejezés egy vagy többszöri előfordulását jelenti.
- Az X* az X nulla vagy többszöri előfordulását jelenti
- Az X? az X nulla vagy egyszeri előfordulását jelenti.
- A +,*,? esetén fontos a csoportosítás, ami sima zárójellel történik, mivel mindhárom a tőle balra található regex csoportra vonatkozik!
- A +,*,? operátorok a lehető legtöbb karaktert illesztik!
  - Pl: A `"\\d+"` az `"123u"` stringen az `"123"`-ra fog illeszkedni, nem csak simán az `"1"`-re
  - Pl: `"\\(.*\\)"` és `"x(abc)y(de)z"` esetén azt hihetnénk, hogy két illeszkedés lesz, az `"(abc)"` és a `"(de)"`, közben pedig csak egy van, az `"(abc)y(de)"`

Reguláris kifejezések

- A `matches()` metódus eldönti, hogy egy reguláris kifejezés illeszkedik-e a karakterláncra.
  - `boolean b = "cdebabbaaabbfgh".matches("cde[ab]*fgh"); // true`
- Pl: találjunk meg olyan `String`-eket, amiben ”://” van. Nézzük meg mondjuk a http://domained.hu –ra (tehát ”http”-re és ”domained.hu”-ra számítunk mint kapott eredmény).
  - regex: ” (.*)://(.*) ” → [\w]+://[\w]+
- Pl: találjunk meg olyan `String`-eket, amiben e-mail cím van. Nézzük meg mondjuk a cimed@domain.hu –ra (tehát ”cimed”-re, ”domain”-re, és ”hu”-ra számítunk, ! ”.”-ra vigyázni!).
  - regex: ” (.*)@(.*)\.(.*) ” → [\w]+@[\w]+\.[\w]+
- Tutorial és referencia:
  - http://docs.oracle.com/javase/tutorial/essential/regex/
  - http://www.regular-expressions.info/reference.html
  - http://www.ocpsoft.org/opensource/guide-to-regular-expressions-in-java-part-1/

Reguláris kifejezések

- A `Pattern`: `java.util.regex.Pattern` osztály, egy reguláris kifejezést (pl.: egy `String`) `Pattern`-be (mintába) fordít át. Rossz szintaxis esetén `PatternSyntaxException`-t kapunk. Pl.:

```
String regex = new String("..");
Pattern p = Pattern.compile(regex);
```

- Ezek után ezt a p-t egy `Matcher`-be kell „tenni” (`java.util.regex.Matcher`), ami tetszőleges karakter-sorozatot vár. A `Pattern`t (mintát) erre az inputra fogja alkalmazni, tehát megnézi, hogy ha rátesszük erre a mintára a bemenetet, mit kapunk eredményül. Tehát:

```
String s = new String("ab");
Matcher m = p.matcher(s);
```

- `Matcher` függvények:
  - `matches()`: egyezés ellenőrzése (illeszkedik-e a mintára a szöveg)
  - `find()`: a következő részsorozatra ugrik, ami megfelel a mintának
  - `group()`: visszaadja az előző részsorozatot, ami megfelelt a találatnak

Reguláris kifejezések

- Nézzük meg a `regex` csomag tartalmát és teszteljük a kifejezéseket!
- `Scanner` osztály: a `java.util.Scanner` osztály primitív típusokat és `String`-et tud tördelni reguláris kifejezések segítségével. Az inputját tokenekbe tördeli, delimiter pattern (sablon) szerint (ez egy speciális osztály, kifejezetten az egyszerű tördelésre reguláris kifejezés segítségével).
  - Tagfüggvényei (például)
    - `nextInt()`,
    - `hasNext()`,
    - `next()`
  - FIGYELEM! Ha a következő token (darabka) más típusú, mint amilyet várunk, akkor `InputMismatchException` váltódik ki.
- Nézzük meg a *SampleScanner.java*-t!

Kivételkezelés I.

- A program futása során előfordulhatnak (előre látható) hibák, többek között a következő okokból kifolyólag:
  - Kódolási hiba: valamit rosszul írtunk meg, így nem az történik, amit elgondoltunk
  - “Külső” hiba: egy általunk igénybe vett szolgáltatás nem működik megfelelően (például szeretnénk írni a HDD-re, de elfogyott a hely)
  - Adathiba: a program számára biztosított bemenő adatok nem felelnek meg az elvártnak (például megszegik az elő specifikációt)
- A futás során tapasztalt hibákat, vagy egyéb speciális állapotokat szeretnénk egységes, átlátható, strukturált módon kezelni
  - A konzolra kiírt szöveges üzenetek a legkevésbé sem teljesítik ezeket a kritériumokat
  - És a hibajelzésre használt visszatérési értékek sem
- Ennek az eszköze a modern nyelvekben a kivételkezelés. Célja:
  - Kis erőfeszítéssel, az olvashatóságot és az eleganciát megtartva lehessen kivételeket kezelni
  - Általában a kivételeket kezelő kódot elválasztja a többitől, a „lényegtől”
  - Az elkészült programok megbízhatóságát, olvashatóságát növeli

Kivételkezelés II.

- A kivétel a programnak az elvárttól eltérő állapotát leíró objektum
- Amennyiben a futás során valahol abnormális állapotot észlelünk, úgy ott egy kivételt “dobunk”, amivel jelezzük a problémát a külvilág (jó esetben a program többi része) felé
  - Például egy adatbeolvasó függvényben, ami csak egész számokat képes feldolgozni, string bemenet esetén dobunk egy kivételt, mivel ez a függvény elő specifikációjának nem felel meg, és így nem vagyunk felkészülve a bemenet értelmezésére
- A kivételbe elhelyezünk minden információt, ami a hiba kezeléséhez szükséges (például, hogy melyik beolvasott file hibás)
- Kivétel != Hiba
  - A kivételek nem mindig hibát jeleznek, lehet, hogy csak egy ritkán bekövetkező, vagy a feladat szempontjából kevésbé fontos eseményt

Kivételkezelés III.

- A dobott kivételeket elkapjuk (ez történhet a program egy egészen távoli pontján is), és megpróbáljuk valamilyen értelmes módon lekezelni (felhasználva a benne kódolt információkat, pl. a hibás file nevét)
- A lekezelés lehet újbóli próbálkozás (pl. kapcsolathiba esetén), a hiba okának megszüntetése (viszonylag ritkán tudjuk megtenni), vagy a hiba rögzítése és jelzése a felhasználónak (hibaüzenet, felugró ablak, stb.)
- A lekezelés során mindig figyelnünk kell arra, hogy az összes objektumot értelmes, konzisztens állapotba juttassuk (azaz megszüntessük az abnormális állapotot)

Kivételek típusai Java-ban

`java.lang.Throwable`:

- Minden kivétel őse, a kiváltható osztályok hierarchiájának csúcsa. Ezen osztály objektumainak és leszármazottainak különlegessége, hogy a `throw` utasítással kiválthatók.

`java.lang.Exception`:

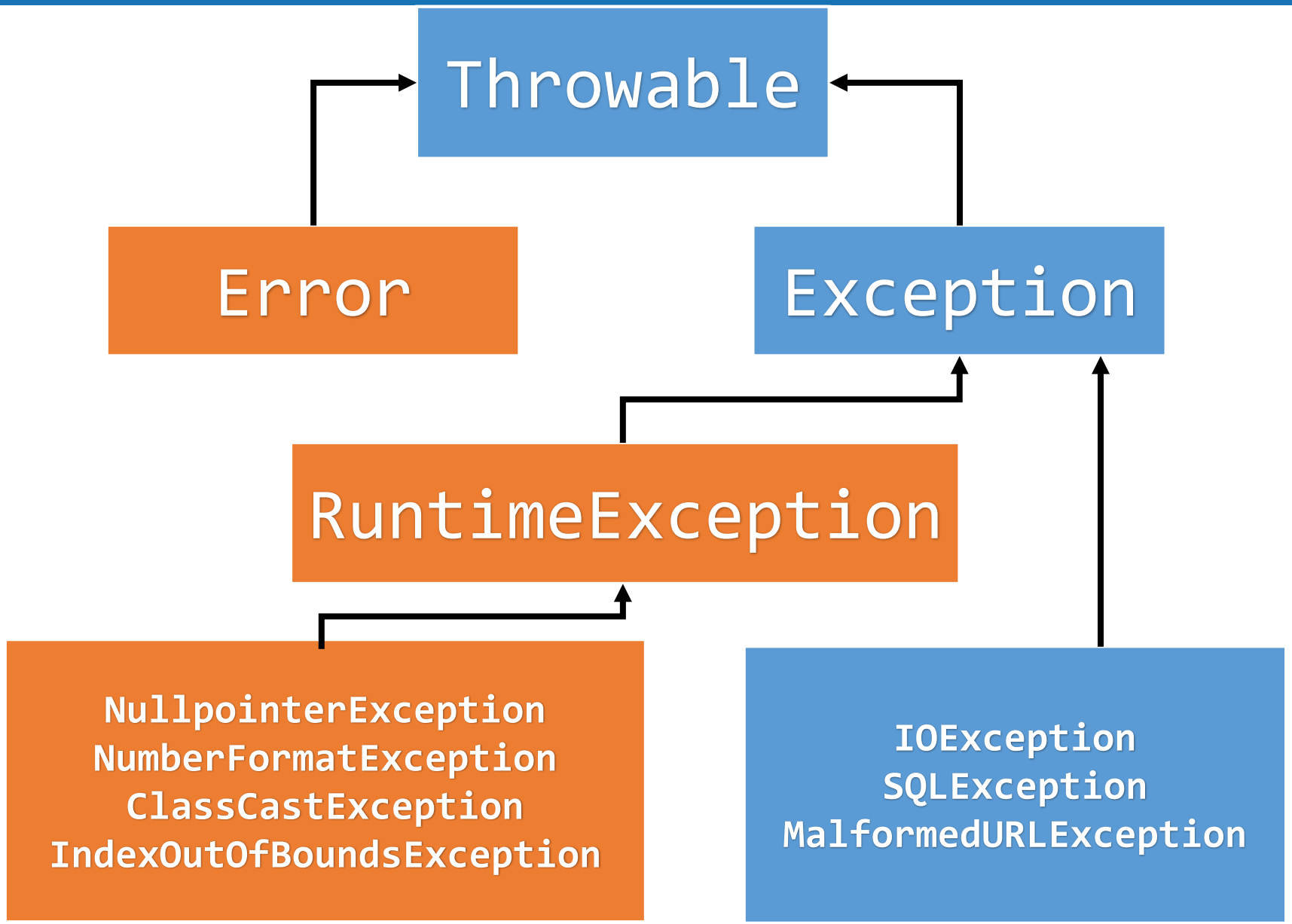
- Ez, és a leszármazottai (a `RuntimeException` és leszármazottai kivételével) ellenőrzött kivételosztályok.
  A program teljes kivételkezelését lényegében az `Exception` és leszármazottai segítségével oldjuk meg. A saját kivétel osztályunkat is az `Exception`ből célszerű (illik és kell) származtatni.

`java.lang.RuntimeException:`

- Futási időben bekövetkező, előre nem látható kivétel
  pl.: nullpointer használata.

`java.lang.Error:`

- Komoly hiba esetén lép fel. Több leszármazottja van. Előre nem látható, futást gátoló problémát jelez, így nem kötelező kezelni.
  Pl. `OutOfMemoryError` (elfogyott a memória).


Throwable
Error
Exception
RuntimeException
NullpointerException
NumberFormatException
ClassCastException
IndexOutOfBoundsException
IOException
SQLException
MalformedURLException

Kivételkezelés Java-ban I.

- Java-ban minden kiváltott kivétel nem csupán jelzés a rendszertől, hanem **konkrét információt is tartalmaz** a kivételről, illetve a kivétel kiváltásának állapotáról (az öröklődésnek köszönhetően a saját kivételosztályainkban tetszőlegesen használhatjuk, elfedhetjük).

- Ezeket a következő függvényekkel érhetjük el:
  - `getMessage()`
  - `getLocalizedMessage()`
  - `getStackTrace()`

Kivételkezelés Java-ban II.

- Kivétel három különböző módon keletkezhet:

1. A program futása közben valamilyen **rendellenes**, előre nem, vagy csak nehezen látható **dolog történik**: pl.: nullával osztás, erőforrások elfogyása, osztálybetöltési hiba, stb. (Nem ellenőrzött kivétel)
2. A programban egy **`throw`** utasítás váltja ki a kivételt, ez lehet a könyvtári csomagban vagy a saját osztályainkban definiált kivétel is.
3. Egy **aszinkron** kivétel lépett fel. Ez akkor következhet be, amikor a program több szálon fut és egy másik szál futása megszakad. (Ezzel most nem foglalkozunk)

Kivételkezelés Java-ban III.

- Az első típus bárhol előfordulhat a programkódban.
- Számunkra a második eset a legérdekesebb, hiszen ezzel saját magunk tudjuk befolyásolni a program futását.

- Az ezen a módon kiváltott kivételt jelezni kell, s e kivételek csak egy jól specifikált helyen, a kivétel hatáskörén belül következhetnek be (**`try` blokk**). A **`try` blokk** figyeli a benne szereplő utasításokat kivétel keletkezése szempontjából.

- Ha egy kivétel dobódik, akkor a programkód futása ott megszakad, azaz a **`try` blokkon** belüli, a kivétel kiváltása utáni utasítások nem fognak lefutni

Kivételkezelés Java-ban IV.

- A `try` blokk után találhatóak a `catch` ágak, melyekből több is lehet egymás után.
  - A `catch` ágak elkapják a paraméterüknek megfelelő típusú kivételt és végrehajtják az ághoz tartozó utasításokat.
  - Egy `catch` blokk egy adott típusú, vagy abból származtatott kivételt tud elkapni, ezért fontos a blokkok sorrendje.
  - A `catch` ágak egymás után kerülnek kiértékelésre. Amennyiben a `try` blokkot követő első `catch` által várt kivétel típusa megegyezik a kiváltott kivétel típusával, úgy az fog lefutni (és a többi nem).
  - Amennyiben az első `catch` típusa **nem kompatibilis**, úgy az (esetlegesen meglévő) második `catch` kerül vizsgálatra, és így tovább.
- Ha a **`try` blokk** után nincs megfelelő típust váró `catch`, úgy a kivétel egy szinttel feljebb kerül.
  - Azaz ha egy m metódusban kivétel lép fel, akkor az azt meghívó metódusban is fellép, azon a ponton, ahol meghívtuk az m metódust, hacsak a kivételt le nem kezeljük
  - Ha nincs felsőbb szint (azaz hívó függvény), úgy a program futása megszakad.

Finally

- Ha kódunk tartalmaz olyan részt, mely a kivételektől függetlenül minden esetben lefut, akkor ezt a `finally` nevű blokkal jelezhetjük a fordító számára.

```
public String readFirstLineFromFile(String path) throws
        IOException {
    BufferedReader br = new BufferedReader(new
                                    FileReader(path));
    try {
        return br.readLine();
    } finally {
        if (br != null) br.close();
    }
}
```

Kivételek specifikálása

- Ha egy kivétel fellép egy metódusban, akkor: vagy le kell kezelni vagy jelezni kell, hogy továbbadhatjuk
- A metódusok specifikációja tartalmazza a metódusban fellépő lehetséges kivételeket
  - A paraméterlista és a törzs között `throws` utasítás

```
public static String readFromKeyboard() throws
IOException {
  /*...*/
}
```

Try-catch blokk - gyorsan

- Jelöljük ki a `try` blokkba foglalandó sorokat, majd a kijelölt szövegre kattintsunk jobb gombbal. Itt:
  - Surround With → Try/catch Block
- Ha csak egy utasítás lehetséges kivételeit szeretnénk lekezelni, elegendő, ha a fordítási hibát jelző pirosan aláhúzott szövegrész felé visszük a kurzort. Ekkor a megjelenő tooltip a következő lehetőségeket ajánlja fel:
  - Surround With Try/catch Block
  - Add throws declaration
  - Ha a sor már egy meglévő try-blokkban szerepel:
    Add catch clause to surrounding try

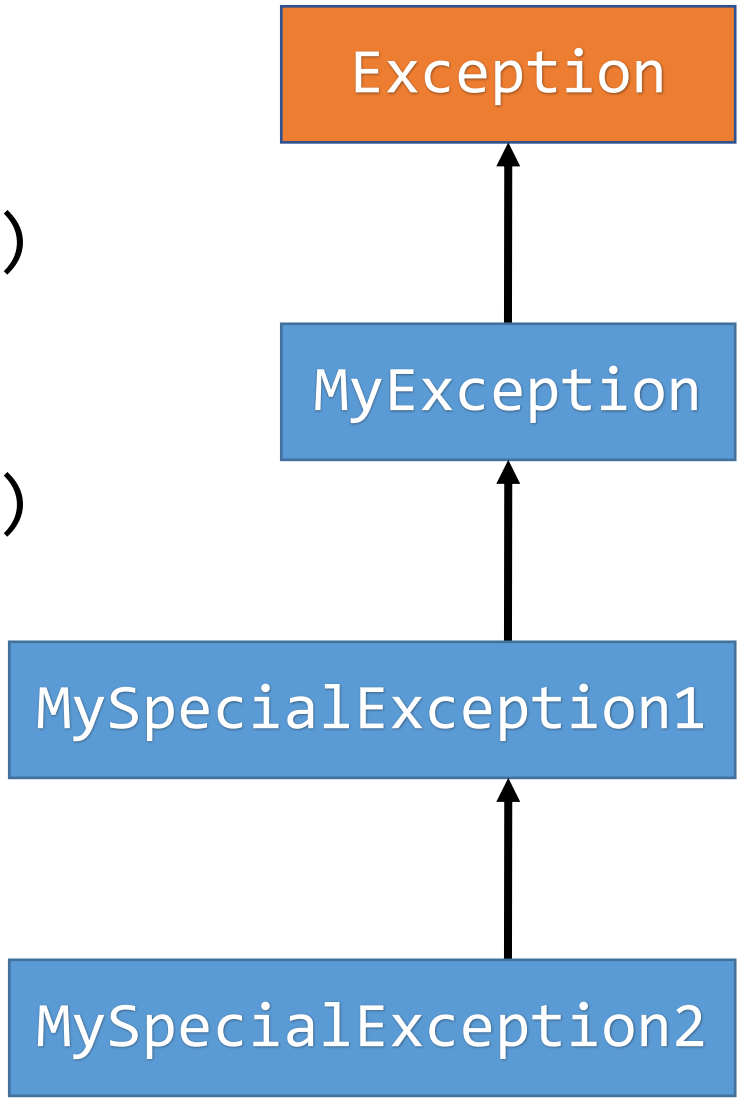
Általános példa

- Sok esetben ajánlott, vagy szükséges saját kivétel létrehozása

```
class MyException extends Exception { /* ... */ }
class ExceptionTest1 {
    public static void main(String[] args) {
        try {
            throw new Exception("Exception");
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
        try {
            throw new MyException("MyException");
        } catch (MyException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Sorrendfüggő példa

```
try {
        /*...*/
}
catch (MySpecialException2 mse2)
{
        /*...*/
}
catch (MySpecialException1 mse1)
{
        /*...*/
}
catch (MyException me) {
        /*...*/
}
catch (Exception e) {
        /*...*/
}
```



try-multi-catch

```
try {
    /* ... */
} catch (IOException | SQLException e) {
    System.err.println(e.getMessage());
}
```

try-with-resources

```
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

```
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br1 = new BufferedReader(new FileReader(path));
         BufferedReader br2 = new BufferedReader(new FileReader(path2)))
    {
        return br2.readLine().append(br1.readLine());
    }
}
```

Assertion

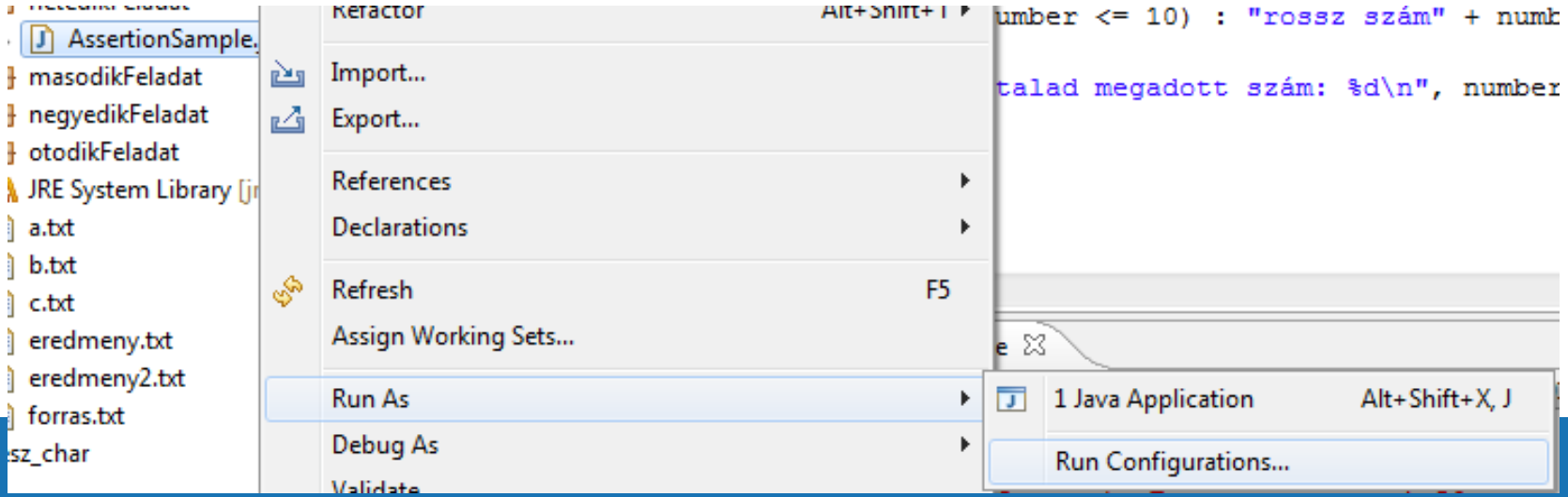
- Amikor implementálunk vagy debuggolunk egy osztályt, akkor sokszor hasznos, ha meg tudunk adni olyan feltételeket, amelyek biztosan igazak egy metódusban. Ezeket a feltételeket assertionnek (követelésnek) hívjuk, ami segít biztosítani egy program helyességét (assertion lehet még pl.: az előfeltétel és az utófeltétel). Tehát az assertion programhelyességi előírást takar, hármas szerepe van:
  - Dokumentációs eszköz: szemantika dokumentálását szolgálja
  - A szemantika absztrakt (formális) leírását tartalmazza
  - A program futása során a bizonyos szemantikai feltételek fennállásának ellenőrzését szolgálja. Ha a feltétel nincs meg, akkor a programnak tudnia kell reagálni erre a szituációra.

Assertion

- A Java `assert` egy logikai típusú kifejezést ellenőriz a program futásidejű végrehajtásánál. Az `AssertionError` osztály fűződik hozzá (így ilyen típusú kivételeket tud dobni ha a kifejezés hamis).
- Kétféle típusa van:
  - `assert expression;`
    - Ekkor kiértékeli az `expression`-t (kifejezést), és `AssertionError` kivételt dob, ha a kifejezés hamis.
  - `assert expression1 : expression2;`
    - Ekkor kiértékeljük az `expression1`-et, és `AssertionError` kivétel dobódik `expression2` hibaüzenettel, ha az `expression1` kifejezés hamis.

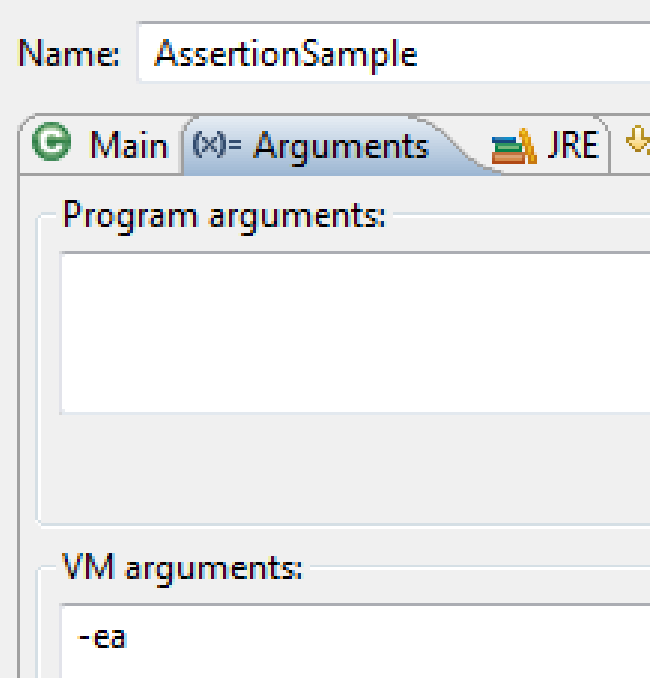
Assertion

- Nézzük meg az 'assertfeladat' csomaghoz tartozó kódot!
- Teszteljük, hogy mi történik, ha a tartományon belül és kívül adunk meg egy számot!
- A Java-ban alapértelmezettként az `assert` le van tiltva, mert lassítja a VM-et. Most viszont kapcsoljuk be, ezt a következőképpen tehetjük meg:
  - Jobb kattintás az AssertionSample-re -> Run As -> Run Configuration



Assertion

- Arguments -> VM arguments, ide írjuk be: -ea



- Most már tartományon kívüli szám esetén kivétel váltódik ki
  - Ismét teszteljük, hogy mi történik, ha a tartományon belül és kívül adunk meg egy számot!

A File osztály

- A `java.io.File` osztály példányai fájlneveket reprezentálnak. Egy ilyen objektum egy elérési utat tartalmaz, ami a fájlrendszer egy fájlját vagy könyvtárát azonosíthatja (ha létezik).
- Konstruktorok:
  - `File f1 = new File("course/week2/25.html");`
  - `File f2 = new File("D:\\public\\html\\course\\week2\\25.htm");`
  - `File f3 = new File("course/week2", "25.html");`
- Lehetőségek:
  - Információk lekérése
    - `exists(), isFile(), getPath(), canWrite(), lastModified()`
  - Filemanipuláció:
    - `createNewFile(), renameTo(), delete(), setReadable()`
  - Könyvtárlétrehozás: `mkdir()`
  - Könyvtárlistázás: `list()`
  - Fájllistázás: `listFiles()` (Listázásnál filtereket adhatunk meg)
  - Nézzük meg a `diskio` csomag `FileSpy` osztályát!

Temp fájlok

- A `File` osztály segítségével létrehozhatunk temporary fájlokat, melyek a program végrehajtása után törlődnek. Ez néha hasznos lehet, hogy ne szennyezzük a gépünket olyan fájlokkal, amelyekre csak a program futása közben van szükségünk, később nem.

```
try {
  // temp fájl létrehozása az alapértelmezett temp könyvtárba.
  // Harmadik paramérterként megadható tetszőleges könyvtár is.
  File temp = File.createTempFile("filename", ".txt");
  // A temp fájl törlése a program végrehajtása után
  temp.deleteOnExit();
  // Írás a temp fájlba
  BufferedWriter out = new BufferedWriter(new FileWriter(temp));
  out.write("…");
  out.close();
} catch (IOException e) { /*...*/ }
```

Gyakorló feladat G02F01

- Írj egy szám és állatkitaláló alkalmazást Java programozási nyelv használatával.
  - A következő csavarral: nem csak számot, hanem állatokat is ki kell találnod!
  - (egy kis könnyítés, csak a 4 lábú haszonállat közül kell találgatni)
  - (Figyelem a macska NEM haszonállat, de a kutya igen!)
  - A program így működjön:
    - 1. szépen üdvözöl
    - 2. megkér, hogy mondj egy számot, meg egy állatot
    - 3. ellenőrizze az eredményt
      - 3. a) ha az állat stimmel (pl. ló), de kevesebbet írtál, ezt írja ki: én több lóra gondoltam!
      - 3. b) ha a szám stimmel, akkor ezt írja ki: Pont ennyire gondoltam, de nem disznóra!
      - 3. c) ha minden stimmel gratuláljon és lépjen ki!
    - 4. mentse el egy fájlba, mind az inputot, mind az outputot! (rendezetten!)
  - A bemenetet minden alkalommal egy `String`nek kell tekinteni, melyből regex-szel keresse ki a programod, hogy mit is akartál.
    - Például:
    - „55 boci”
    - „56 boci”
  - A program 1 és 101 közötti csordára gondolhat!
    - Nem csak 4 lábú haszonállat, hanem az összes!
    - Oldd meg, hogy a program többször gondoljon szárnyas állatra, mint sem (ne teljesen random legyen az állatkitalálós része, hanem „szeresse” jobban a szárnyasokat!)
    - Amikor tippelek, és pl. 10-zel többet tippelek, akkor ezt írja ki: „Sokkal kevesebbre gondoltam!”

Gyakorló feladat G02F02

- Készítsünk programot ami a konzolról beolvas szavakat, ad nekik egy sorszámot, majd kiírja a szónak megfelelő nevű fájlba. Ha a begépelt szó „exit” akkor kilép.
- (Tehát ha az első beolvasott szó az volt, hogy macska akkor létrehoz egy macska.txt nevű fájlt aminek a tartalma „1 macska” lesz, ha a második szó „kutya” volt akkor létrehoz egy kutya.txt nevű fájlt aminek a tartalma „2 kutya” lesz. stb.)
- Ha ezzel megvagyunk készítsünk egy menüt, ahol az 1-es gomb megnyomásával az előbb megirt programot lehet futtatni (az ha végzett ide térjen vissza), a 2-es megnyomásával az órán megírt fájl összemásolót!

Gyakorló feladat G02F03

- Készítsünk programot aminek van menüje, 4 funkcióval. Törekedjünk tájékoztatni a felhasználót, hogy mi történt!
- Az első funkció, hogy meg lehessen adni egy fájlt, amit megnyit írásra. Ekkor begépelvén szavakat a konzolra hozzáírja a már a fájlban lévő adatokhoz mindig új sorba. Az "exit" szóra lehessen visszatérni a menübe.
- A második funkció, hogy megadnunk egy fájlt (pl. az első funkcióval generáltat). Ekkor irassa ki a megadott fájlból a felhasználó által megadott mintára illeszkedő sorokat, és hogy az adott sorok hanyadikként helyezkednek el a fájlban.
- Harmadik funkció: a megadott fájl esetén számláld meg a fájl sorait, a fájlban lévő szavak számát, valamint a karakterek számát, majd az eredményeket irasd ki egy fájlba, melynek fájlneve: eredetifilenev_count.txt (értelemszerűen behelyettesítve az eredeti fájlnevet).
- Negyedik funkció: kilépés.
- Ne feledkezz meg az esetlegesen keletkező exception-ök kezeléséről! (mintaillesztés, fájl-megnyitás, egyéb I/O hiba estén)

Gyakorló feladat G02F04

- Készíts programot, mely HTML fájlokat dolgoz fel, illetve készít, és amelyiknek az egyes funkcióit menü segítségével lehet kiválasztani. Törekedjünk tájékoztatni a felhasználót, hogy mi történt!
- Az első funkció egy nagyon egyszerű HTML fájl készítése. A felhasználó megadja a készítendő fájl nevét, a HTML oldal címét, és az oldal szövegét, ezek alapján létrehozzuk a fájlt.
- A második funkció, hogy egy, a felhasználó által megadott HTML fájlt megfosztunk minden HTML tagtől (minden olyan szöveg ami < és > között van) és az így kapott szöveget elmentjük az eredeti fájllal megegyező nevű, de .txt kiterjesztésű fájlba. Ezt a feladatot reguláris kifejezések segítségével oldd meg!
- A harmadik funkció, hogy a program megszámolja egy megadott HTML fájl sorainak és karaktereinek számát. Az eredményt a java.util.Formatter segítségével írd ki a konzolra, a paraméterek az oldal címe (ami a <title> és </title> tagek között szerepel), és a két megszámolt érték legyenek.
- Formatter API: http://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html
- Negyedik funkció: kilépés
- Ne feledkezz meg az esetlegesen keletkező exception-ök kezeléséről! (pl. fájlmegnyitás, egyéb I/O hiba estén). Ha a felhasználó nem HTML fájlt ad meg (rossz kiterjesztés), vagy nem jó HTML fájlt ad meg, a program tájékoztassa a felhasználót a hibáról, majd lépjen ki. Ezt is kivételkezeléssel oldd meg!
    - A HTML fájl helyességének vizsgálatakor elegendő a következő dián található példában szereplő tagekre ellenőrizni, hogy szerepelnek-e a nyitó és záró tagek is, és hogy a záró tag a nyitó tag után szerepel-e.
- A programot teszteld tetszőleges egyszerűbb HTML fájlon, vagy a gyakorlati anyagban megtalálható index.html fájlon. A második funkciónál elegendő, ha a program az ebben a fájlban található tagekre helyesen működik.

Gyakorló feladat G02F05

- Írj egy alkalmazottakat nyilvántartó alkalmazást Java programozási nyelv használatával.
- Egy alkalmazott adatai a következők:
  - Név
  - Hálózati azonosító, (hat karakter és két szám)
  - E-mail
  - Telefonszám (bármely mobil az alábbi formátumban: +36-XX/XXX-XXXX)
  - Pl.: Kiss István;kisist01;kiss.istvan@ceg.hu;+36-30/555-5555
- A programnak két funkciója van:
  - 1. Keresés az alkalmazottak között (azonosító alapján)
  - 2. Új alkalmazott regisztrálása a rendszerbe
    - A hálózati azonosítót azonosítóját generálja a program a név alapján!
      - Vezetéknév első 3 betűje, keresztnév első 3 betűje és egy véletlen szám, ami kétjegyű.
    - Minden mező tartalmát ellenőrizd! Ha nem valid a mező tartalma, jelezd a felhasználó felé (minél jobban segítve a megfelelő bevitelt), és kérd be újra az adatot!
    - A név esetében ellenőrizze a nagybetűhelyességet, és automatikusan formázza meg az alkalmazás!
- Az alkalmazás fájlban tárolja a regisztrált alkalmazottakat
  - Program indulásával töltődjön be a fájl, és utána keressen az adatok között
    - Tipp:
      - A program futása elején foglaljatok le fix 100 elemű tömböt.
      - Ha ettől több adatot kéne beolvasni (kapott fájl több sort tartalmaz), álljon le a beolvasás, és csak az első 250 alkalmazott adatait olvassa be
      - Új alkalmazott regisztrálásánál CSAK akkor vegye fel, a alkalmazottat és írja ki a fájlba, ha még a alkalmazott száma nem érte el a 250-et!
      - Magyarul 250 alkalmazott fölött értesítsétek a felhasználót, és ne történjen változás az adatbázisban.
  - Új alkalmazott regisztrálásakor automatikusan adja hozzá a fájlhoz az adatokat (fűzze a végére)
  - Fájl neve cegek, kiterjesztése .ceg legyen (alkalmazott.ceg)
  - Fájlformátum: egy alkalmazott adatai egy sorban legyenek (minden alkalmazott külön sor)
    - név;azonosító;email;telefon
    - Az adatok NEM tartalmazhatnak ;-t, az az adatok elválasztására van fenntartva!!!

Gyakorló feladat G02F06

- Írj egy fájlokkal dolgozó alkalmazást, mely a következő feladatokat látja el:
  - Main osztály (egyetlen main metódussal, ez az alkalmazás „váza”)
    - Gyakorlatilag egy konzolos menü, ahol a felhasználó választhat a meglévő funkciók közül
  - Választható funkciók: (legyenek a Main-től független Functions osztály metódusai)
    - 1 – szövegfájl statisztika (bekéri a felhasználótól a fájl nevét, majd ha az egy szöveges fájl, kiírja a fájl sorainak, szavainak és karaktereinek számát szóközzel és szóköz nélkül is)
    - 2 – szövegfájl másoló (bekéri a felhasználótól a forrás és cél fájl neveit, majd a forrás fájl tartalmát átmásolja a cél fájlba – kezelje le, hogy mi történik, ha már létezik a célfájl a következő három lehetőség közül: felülírás, hozzáfűzés, hibajelzés; itt a programozó döntésére van bízva, hogy mit választ)
    - 3 – szövegfájl kereső (bekéri a felhasználótól a fájl nevét és a keresendő szót, kifejezést, majd kiírja, hogy hányszor szerepelt a fájlban, találat esetén kilistázza a találatokat is aszerint, hogy hányadik sorban hányadik szó a keresett szó) – elegendő teljes keresést végezni, de lehet kísérletezni is, hátha van hatékonyabb megoldás
    - 0 – kilépés (leállítja az alkalmazást – a funkció meghívásáig folyamatosan fusson az alkalmazás, egy funkció végrehajtása után újra jelenjen meg a menü és várjon a következő parancs kiadásáig)
  - Elegendő txt fájlokkal dolgozni, de a felhasználó felé minden esetben jelezni kell, ha a fájl nem megfelelő formátumú! Ügyelj a megfelelő kivételkezelésre, és a be és kimenetek lezárására!

Gyakorló feladat G02F07

- Írj egy csoport hallgatóit nyilvántartó alkalmazást Java programozási nyelv használatával.
- Egy hallgató adatai a következők:
  - Név
  - Hálózati azonosító, (négy karakter és két szám)
  - E-mail
  - Képzési kód
  - Pl.: Kiss István;kiis01;kiss.istvan@egyetem.hu;MI-BSc
- A programnak két funkciója van:
  - 1. Keresés a hallgatók között (azonosító, vagy név alapján)
    - A találatokat szépen formázva írja ki a konzolra
  - 2. Új hallgató regisztrálása a rendszerbe
    - A hálózati azonosítóját generálja a program a név alapján!
      - Vezetéknév első 2 betűje, keresztnév első 2 betűje és egy véletlen szám, ami kétjegyű.
    - Minden mező tartalmát ellenőrizd! Ha nem valid a mező tartalma, jelezd a felhasználó felé (minél jobban segítve a megfelelő bevitelt), és kérd be újra az adatot!
    - A név esetében ellenőrizze a nagybetűhelyességet, és automatikusan formázza meg az alkalmazás!

Gyakorló feladat G02F07

- Az alkalmazás fájlban tárolja a regisztrált hallgatókat
  - Program indulásával töltődjön be a fájl, és utána keressen az adatok között
    - Tipp:
      - A program futása elején foglaljatok le fix 40 elemű tömböt.
      - Ha ennél több adatot kéne beolvasni (kapott fájl több sort tartalmaz), álljon le a beolvasás, és csak az első 40 hallgató adatait olvassa be
      - Új hallgató regisztrálásánál CSAK akkor vegye fel a hallgatót és írja ki a fájlba, ha a hallgatók száma még nem érte el a 40-et!
      - Magyarul 40 hallgató fölött értesítsétek a felhasználót, és ne történjen változás az adatbázisban.
  - Új hallgató regisztrálásakor automatikusan adja hozzá a fájlhoz az adatokat (fűzze a végére)
  - Fájl neve hallgatok, kiterjesztése .csp legyen (hallgatok.csp)
  - Fájlformátum: egy hallgató adatai egy sorban legyenek (minden hallgató külön sor)
    - név;azonosító;email;képzési kód
    - Az adatok NEM tartalmazhatnak ;-t, az az adatok elválasztására van fenntartva!!!

Gyakorló feladat G02F08

- Írj egy olyan programot, ami egy egyszerű konzolos menü segítségével old meg speciális fájlkezelési feladatokat, .txt fájlokra, a következők szerint:
  - Legyen egy Main osztály (egyetlen main metódussal, ez az alkalmazás „váza”)
    - Gyakorlatilag egy konzolos menü, ahol a felhasználó választhat a meglévő funkciók közül
    - A program indulásakor kérjen be egy könyvtár elérési utat a konzolról, ahol a feldolgozandó szöveges fájlok találhatók, valamint egy regex-et, ami alapján a fájlokat fogjuk szűrni a megadott mappában.
  - Választható funkciók: (legyenek a Main-től független Functions osztály metódusai)
    - 1 – Listázza ki az előzőleg bekért mappában található fájlokból azokat, aminek a neve illeszkedik a regex-re. A listát mentse el a list.txt fájlba úgy, hogy soronként egy fájlnevet tartalmazzon.
    - 2 – Fűzze össze a fájlokat, és tárolja el az output.txt-ben.
    - 3 – Kérjen be egy másik regex-et, amivel szavakat keresünk a fájlokban. Írja ki, hogy melyik fájlban hányszor szerepelt a regex-nek megfelelő szó „táblázatos” formában, azaz a fájlnév után megfelelő távolságra írja ki a számot. Mentse is el ilyen formában, a stat.txt fájlba.
      Pl: alma.txt 5
      kortefa.txt 21
    - 0 – kilépés (leállítja az alkalmazást – a funkció meghívásáig folyamatosan fusson az alkalmazás, egy funkció végrehajtása után újra jelenjen meg a menü és várjon a következő parancs kiadásáig)
  - Elegendő txt fájlokkal dolgozni, de a felhasználó felé minden esetben jelezni kell, ha a fájl nem megfelelő formátumú! Ügyelj a megfelelő kivételkezelésre, és a be és kimenetek lezárására!
  - A wiki-n található fájl csomagon könnyen tesztelheted a működést.