

Tóth József

**Asztali alkalmazások fejlesztése II.**

Tömbök, listák, fájlok, osztályok

# Tartalom

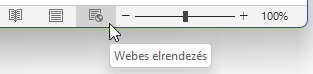
[Útmutató](#_Útmutató)

1. [Tömbök 1.](#_01._Tömbök_1._1)  
   Tömbök létrehozása, használata, kockadobás statisztika
2. [Tömbök 2.](#_02._Tömbök_2.)  
   length, lottószámok sorsolása, kezdőértékek megadása, római számok
3. [Tömbök 3.](#_03._Tömbök_3.)  
   Fordítós játék
4. [Tömbök 4.](#_04._Tömbök_4.)  
   Hőmérsékletek 1: átlag, minimum, maximum
5. [Tömbök 5.](#_05._Tömbök_5.)  
   Hőmérsékletek 2: leghidegebb/legmelegebb nap, fagyos napok száma, módosítás, diagram
6. [Tömbök 6.](#_06._Rendezés)  
   Buborékos rendezés (növekvő/csökkenő, stringek: compareTo), bolygók
7. [Tömbök 7.](#_07._Tömbök_7.)  
   Vetélkedő, Arrays osztály, csomagoló osztályok, csökkenő rendezés
8. [Tömbök 8.](#_08._Hegyek)  
   Hegyek feladat
9. [Tömbök 9.](#_09._Kétdimenziós_tömbök)  
   Kétdimenziós tömbök, dobás 2 kockával
10. [Tömbök 10.](#_10._Népesség)  
    Népesség feladat
11. [Listák 1.](#_11._Listák_1.)  
    ArrayList használata (add, set, get, remove, size), Todo program, Integer.parseInt
12. [Listák 2.](#_12._Listák_2.)  
    ArrayList használata (subList, másolat készítése), 3 legnagyobb felajánlás
13. [Fájlok 1.](#_13._Fájlok_1.)  
    Kivételek, NumberFormatException, szöveges fájl beolvasása (Scanner), kódolás, hibakezelés
14. [Fájlok 2.](#_14._Fájlok_2.)  
    Írás szöveges fájlba (PrintWriter), hozzáfűzés, egyszerű szövegszerkesztő
15. [Fájlok 3.](#_15._Szavak)   
    Szavak feladat
16. [Fájlok 4.](#_16._Fájlok_4.)  
    Robot feladat, példány létrehozása
17. [Fájlok 5.](#_17._Fájlok_5.)  
    Caesar kódolás, replaceAll, osztály tesztelése
18. [Fájlok 6.](#_18._Fájlok_6.)  
    Dátum átalakítása, split
19. [Fájlok 7.](file:///F:\Szoftverfejleszto\Tananyagok\Java\Java%202\19%20Fájlok%207\19%20Fájlok%207.docx)  
    Hatos lottó statisztika, beolvasás kétdimenziós tömbbe, ellenőrzés debuggerrel
20. [Fájlok 8.](#_20._Fájlok_8.)  
    Olimpiák helyszínei, beolvasás listába, keresés listában, rendezés
21. [Osztályok 1.](#_21._Osztályok_1.)  
    Osztály készítése, getter, konstruktor, példány létrehozása, toString felülírása; Városok távolsága program
22. [Osztályok 2.](#_22._Osztályok_2.)  
    Feliratok 1.
23. [Osztályok 3.](#_23._Osztályok_3.)  
    Feliratok 2., dokumentáció készítése
24. [Osztályok 4.](#_24._Osztályok_4.)  
    Feliratok 3.
25. [Osztályok 5.](#_25._Osztályok_5.)  
    Kéktúra 1., ? operátor, (split)
26. [Osztályok 6.](#_26._Osztályok_6.)  
    Kéktúra 2., Lambda kifejezések, objektumok rendezése
27. [Osztályok 7.](#_27._Osztályok_7.)  
    Kéktúra 3.
28. [Osztályok 8.](#_28._Osztályok_8.)  
    Ki mit tud?
29. [Osztályok 9.](#_29._Osztályok_9.)  
    Színház

[Források](#_Források)

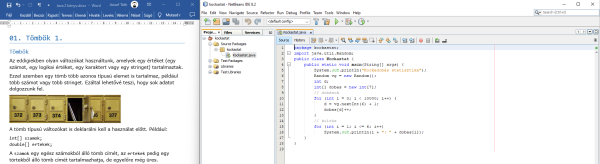
# Útmutató

A könyvet a Microsoft Word webes elrendezés nézetében célszerű olvasni. Ezt a Word ablakának jobb alsó sarkában választhatod ki:

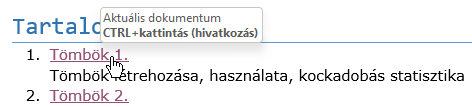


A szöveget nem tördeltem oldalakra, ezért a nyomtatási elrendezésben nehezebben olvasható, és kinyomtatni sem érdemes.

A könnyebb munka érdekében javasolt a Word és a fejlesztőrendszer (NetBeans) egymás mellé rendezése a képernyőn. A szöveg olvasásához általában elég a képernyő szélességének harmada.

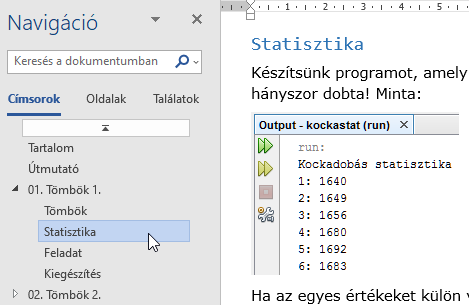


Ha a CTRL billentyűt lenyomva tartva kattintasz a tartalomjegyzékben lévő hivatkozásokra, akkor a megfelelő fejezetre ugorhatsz. Utána az ALT+BALRA NYÍL billentyűk megnyomásával tudsz visszaugrani a tartalomjegyzékhez.



A szövegben történő navigációhoz használhatod a Navigáció ablakot is. Ezt legegyszerűbben az ALT+F billentyűk megnyomásával jelenítheted meg, vagy a Nézet szalagon is bekapcsolható.

A Navigációs ablakban kereshetsz szavakra, vagy a Címsorokat választva egy kattintással a szöveg tetszőleges részére ugorhatsz.



Ha nincs szükséged a Navigációs ablakra, célszerű bezárni, hogy több hely maradjon a szövegnek.

Jó tanulást! 😊

# 01. Tömbök 1.

## Tömbök

Az eddigiekben olyan változókat használtunk, amelyek egy értéket (egy számot, egy logikai értéket, egy karaktert vagy egy stringet) tartalmaztak.

Ezzel szemben egy tömb több azonos típusú elemet is tartalmaz, például több számot vagy több stringet. Ezáltal lehetővé teszi, hogy sok adatot dolgozzunk fel.



A tömb típusú változókat is deklarálni kell a használat előtt. Például:

int[] szamok;  
double[] ertekek;

A szamok egy egész számokból álló tömb címét, az ertekek pedig egy törtekből álló tömb címét tartalmazhatja, de egyelőre még üres.

Ezután létre kell hozni a tömböket, és hozzárendelni őket a változókhoz (inicializálás):

szamok = new int[4];  
ertekek = new double[5+5];

Az első utasítás létrehoz egy 4 egész számból álló tömböt, és a címét eltárolja a szamok változóban.

A második utasítás egy 10 törtszámból álló tömböt hoz létre, amelynek címe az ertekek változóba kerül.

A tömb méretét olyan kifejezéssel kell megadni, amelynek eredménye egy pozitív egész szám.

Ugyanúgy, mint az egyszerű változóknál, a deklaráció és az inicializálás egy sorban is történhet. Például így is csinálhattuk volna:

int[] szamok = new int[4];  
double[] ertekek = new double[5+5];

Egy egész számokból álló tömb létrehozásakor minden eleme 0 lesz. Az alábbi ábra mutatja a szamok tömböt a létrehozás után:



A nyíl azt jelzi, hogy a szamok változó csak a tömb címét tartalmazza. Az ilyen változókat nevezik referencia változónak. (Más objektumoknál, például a stringeknél is hasonló a helyzet. Egy String típusú változó csak a String címét tárolja, tehát szintén referencia változó.)

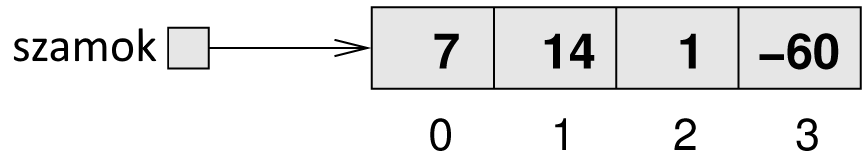
Az ábrán a téglalapokban lévő számok a tömb elemei, az alattuk lévő számok pedig a tömb indexei. Az indexek itt is 0-val kezdődnek, mint a stringeknél. A tömb egyes elemeire úgy hivatkozhatunk, hogy a tömb neve után szögletes zárójelek között megadjuk az indexet. Például a következő utasítás kiírja a tömb nulladik elemét:

System.out.println("A nulladik elem: " + szamok[0]);

A tömb elemei külön-külön is módosíthatók. Például:

szamok[0] = 7;  
szamok[1] = szamok[0] \*2;  
szamok[2]++;  
szamok[3] -= 60;

Nem kell sorban haladni, ezeknek az utasításoknak a sorrendje tetszőleges. Az eredményüket az alábbi ábra mutatja:

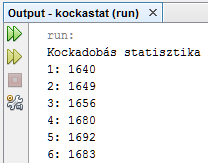


Ha egy tömb minden elemén szeretnénk végrehajtani utasításokat, akkor egy for ciklussal végigmegyünk a tömb elemein. Például, ha ki szeretnénk íratni a szamok tömb elemeit, akkor így csináljuk:

for (int i = 0; i < 4; i++)  
 System.out.println(szamok[i]);

## Statisztika

Készítsünk programot, amely 10 000-szer dob egy dobókockával, majd kiírja, hogy melyik értéket hányszor dobta! Minta:



Ha az egyes értékeket külön változókban tárolnánk (pl. az egyes nevű változó tartalmazná az egyes dobások számát, a kettes a kettesekét, …), akkor mindent 6-szor kellene elvégeznünk. Sokkal jobban járunk, ha egy tömböt alkalmazunk.

A tömbnek 6 eleműnek kellene lennie, de ott a nulladik elem is, bár azt nem szeretnénk használni. Ezért a tömbünk 7 elemű lesz, de a nulladik elemet figyelmen kívül hagyjuk.

A dobas tömb létrehozásakor 7 db 0 értéket fog tartalmazni, és minden dobás után hozzáadunk 1-et a megfelelő elemhez. Ha például 6-ost dobunk, akkor a 6-os indexűhöz adunk egyet. A program végén kiíratjuk a tömb elemeit.

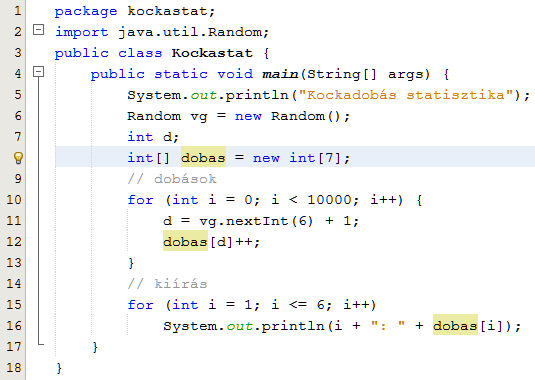


Mit fog tartalmazni a dobas[6] a program végén?   
Írd ide:

A program az inicializálások után két dolgot fog csinálni:

1. "Dob" 10 000-szer a kockával, és minden dobásnál megnöveli a megfelelő tömbelemet 1-gyel.
2. Végigmegy a tömb elemein, és kiírja őket.

Nézzük a programot:



A dobas tömböt a 8. sorban hozzuk létre. A 12. sorban megnöveljük a kisorsolt elemet eggyel. A 16. sorban pedig kiíratjuk az i-dik elemét (az 1-től kezdve!).

Készítsd el és próbáld ki a programot! Figyeld meg, hogy minden futásnál más eredményeket kapsz!

## Feladat

Gondold végig, hogyan lehetne tömb használata nélkül megoldani a feladatot!

## Kiegészítés

A tömbök megadásánál a kapcsos zárójeleket nem csak a típus neve után, hanem a változó neve után is írhatjuk.

Például:

int szamok[] = new int[4];  
double ertekek[] = new double[5+5];

Ez ugyanazt jelenti, mintha ezt írtuk volna:

int[] szamok = new int[4];  
double[] ertekek = new double[5+5];

# 02. Tömbök 2.

## Tömbök mérete

A múltkori programban beírtuk a for ciklusba a tömb méretét. Ez azonban nem túl jó megoldás, mert ha módosítunk a tömb méretén, a for ciklusok határait is át kell írnunk.

Sokkal elegánsabb, ha lekérdezzük a tömb méretét, és így a programok módosítása is egyszerűbb.

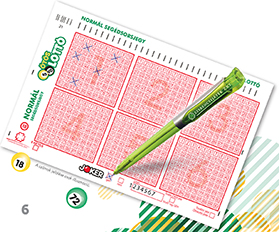
A Java-ban minden tömbnek van egy length nevű változója, amely a tömb méretét, vagyis elemeinek számát tartalmazza.

Például:

int[] t = new int[10];  
for (int i = 0; i < t.length; i++)  
 System.out.println(t[i]);

Ez nagyon hasonlít a stringek length() metódusára, de itt nem teszünk zárójeleket utána, mert ez nem metódus!

## Lottószámok



Szeretnénk az ötös lottón játszani, de nincs ötletünk, hogy milyen számokat jelöljünk be. Most egy olyan programot fogunk írni, amely segít nekünk ebben. Ahányszor lefuttatjuk, mindig kiír öt darab 1 és 90 közötti véletlen számot, növekvő sorrendbe rendezve.

Első nekifutásra megpróbálhatnánk úgy megoldani a feladatot, hogy egy ciklussal előállítunk 5 darab 1 és 90 közötti véletlen egész számot, és kiírjuk őket. Így azonban két probléma is adódna:

1. Előfordulhatna, hogy kétszer (vagy többször) ugyanazt a számot kapjuk.
2. Utána növekvő sorrendbe kellene rendezni őket.

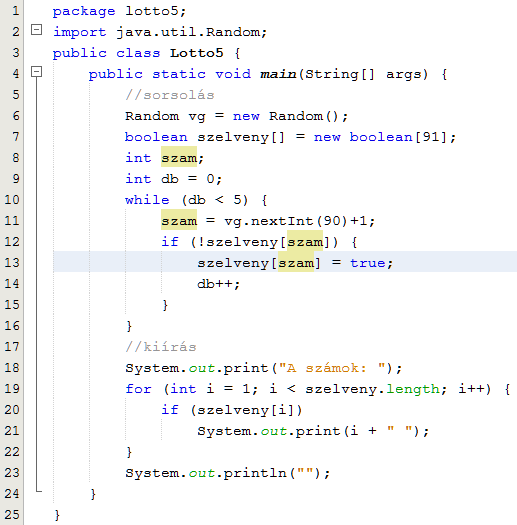
Ezért most más módszert választunk. A lottószelvényt egy tömbbel modellezzük, amelynek elemei boolean típusúak. Minden számnak egy tömbelem felel meg, amely azt jelzi, hogy a szám be van-e jelölve.

Induláskor a tömb minden eleme hamis (false) lesz, ami azt jelenti, hogy még egyik sincs bejelölve. (Ez a Java-ban automatikusan így van.)

Utána előállítunk 1 és 90 közötti véletlen számokat. Ha a tömbben a kisorsolt indexű elem még hamis, akkor igazra (true) állítjuk. Ezzel bejelöljük. Ha már be volt jelölve, újabb számmal próbálkozunk. Így biztosan nem lesz kétszer ugyanaz a szám. Mindezt addig folytatjuk, amíg be nem jelöljük mind az 5 számot.

Végül sorban végigmegyünk a tömb elemein, és amelyik be van jelölve, annak kiíratjuk az indexét.

Nézzük a programot:



A szelveny tömbnek eggyel több (91) eleme van, mert a nulladik elemet most sem használjuk. A kiírást is az 1-es elemnél kezdjük.

Készítsd el és próbáld ki a programot!



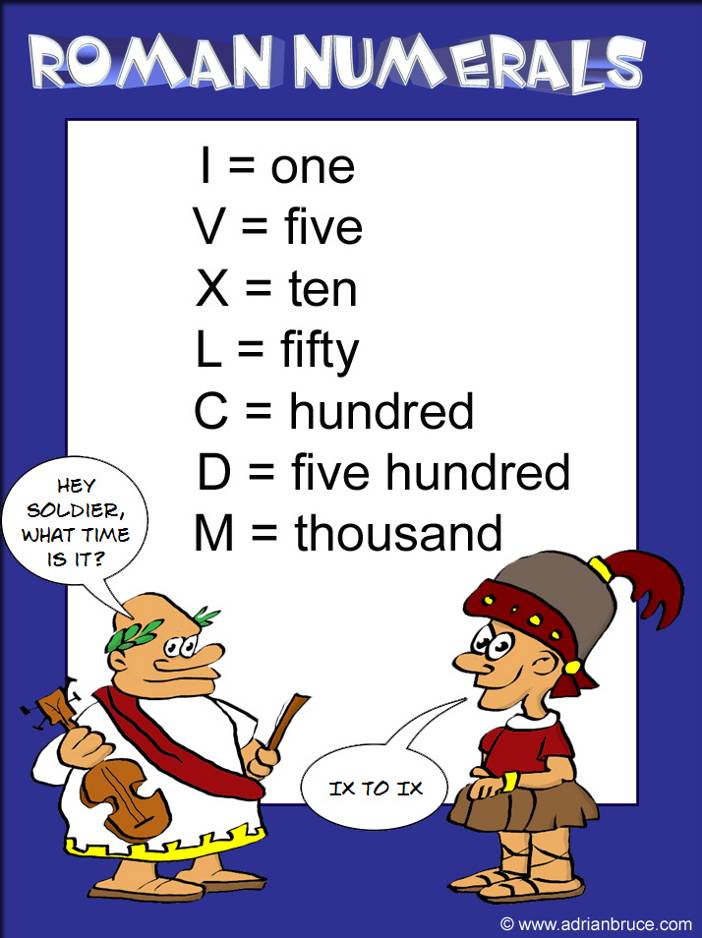
## Tömb kezdőértékeinek megadása

A tömb létrehozásakor rögtön megadhatjuk a tömb elemeit is a következő formában:

int[] szamok = {1,3,5,7,9};

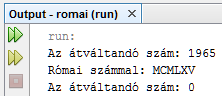
Ezzel létrehozunk egy öt egész számból álló tömböt, amely a felsorolt elemekből áll, és a tömb címét elhelyeztük a szamok változóban. Más típusoknál is lehet alkalmazni. Ezt fogjuk használni a következő példában is.

## Római számok

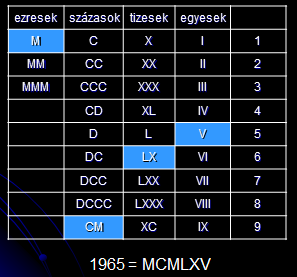


Készítsünk programot, amely a beírt 1 és 3999 közötti egész számot átváltja római számra! A programból a tartományon kívüli számmal (pl. 0-val vagy 5000-rel) lehessen kilépni!

Minta:



Az átváltás elvégzését 1965 esetén az alábbi táblázat szemlélteti:



Vesszük az első számjegyet: 1-es. Az ezresek oszlopból vesszük az első sort: M

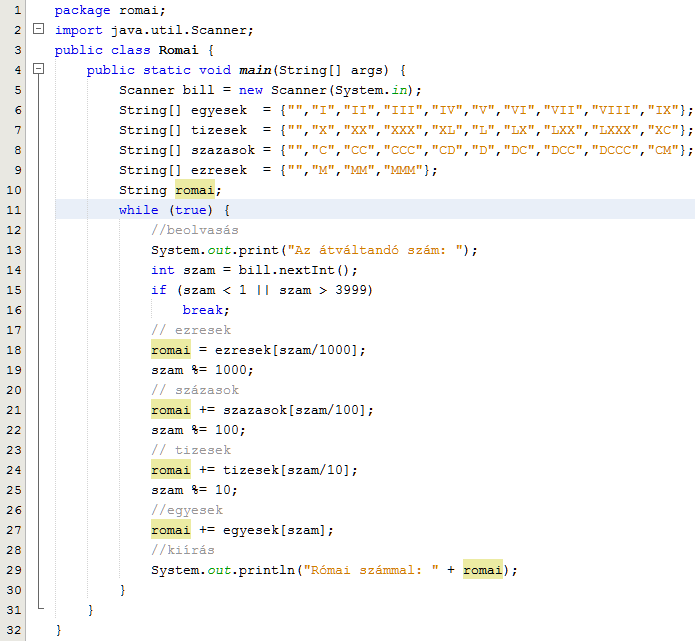
Vesszük a második számjegyet: 9-es. A százasok oszlopban a 9-es sor tartalma: CM

Vesszük a harmadik számjegyet: 6-os. A tizesek oszlopban a 6-os sor tartalma: LX

Vesszük az utolsó számjegyet: 5-ös. Az egyesek oszlopban az 5-ös sorban V van.

Összeolvasva: MCMLXV

Számítsd át 1848-at római számra!   
Írd ide az eredményt:

Nézzük a programot:  


A fenti táblázat elemeit tömbökben tároljuk (6-9. sor). A számjegyeket az egész osztás és a maradékképzés segítségével határozzuk meg (18-28. sor). (Hasonlít a korábban készített címletezéshez.)

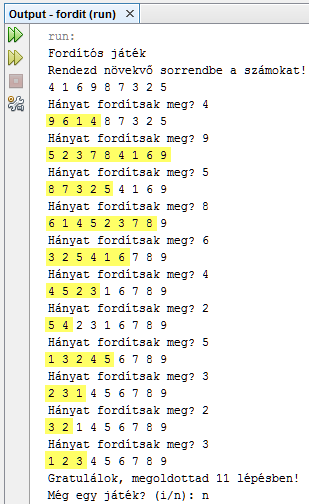
Gondold végig a program működését, készítsd el, majd próbáld ki! Ellenőrizd 1848 átváltását! Próbáld ki érvénytelen évszámmal is!

# 03. Tömbök 3.

## Fordítós játék

A tömbökről tanultak felhasználásával most egy egyszerű logikai játékot fogunk készíteni. A játék célja, hogy minél kevesebb lépésben növekvő sorrendbe rendezzük az induláskor összekevert 1…9 számokat. Minden lépésben megfordíthatjuk az első n elem sorrendjét a listában (n = 2…9).

Minta:



Figyeld meg, hogy a lépések mindig a lista elején fordítják meg az adott számú elem sorrendjét!

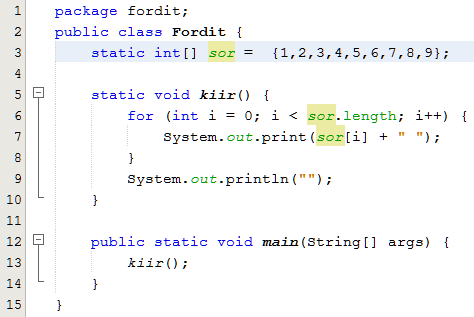
Próbáld ki a kész programot a kapott fordit.jar segítségével! A program futtatásához nyiss egy parancsablakot, válts a .jar fájl mappájába, majd add ki a java -jar fordit.jar parancsot!

Mivel ez a program egy kicsit összetettebb az eddigieknél, először egy egyszerű változatot készítünk, majd azt fokozatosan bővítjük, amíg teljesen készen nem lesz.

## Sor kiírása

A programban szükségünk lesz egy tömbre, amely a számsor elemeit tartalmazza. Mivel több metódusban is fogjuk használni, az osztály elején, a metódusokon kívül fogjuk létrehozni.

A program első változata mindössze ennek a tömbnek a tartalmát fogja kiírni egy kiir() metódus segítségével:



Készítsd el és próbáld ki!



## Elemek megfordítása

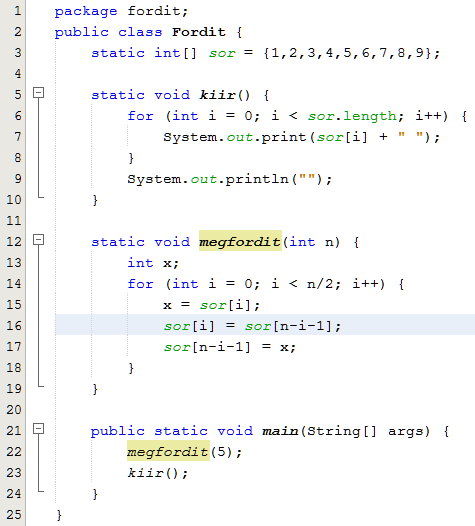
A következő változatban készítünk egy metódust, amely a tömb első n elemét fordítja meg, és ezt is teszteljük a főprogramban.

Hogyan lehet az első n elemet megfordítani? Egy ciklussal végigmegyünk az első n/2 db elemen, és mindig megcseréljük az i-dik és az n–i–1–dik elemet.

A következő táblázat n=5 esetén mutatja a cseréket. i értéke 0, 1 és 2 lesz. n–i–1 értéke rendre 4, 3 és 2.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i=0 | 5 | 2 | 3 | 4 | 1 | 6 | 7 | 8 | 9 |
| i=1 | 5 | 4 | 3 | 2 | 1 | 6 | 7 | 8 | 9 |
| i=2 | 5 | 4 | 3 | 2 | 1 | 6 | 7 | 8 | 9 |

Nézzük a megoldást:



Az elemek cseréje a 14-17. sorban történik. A cseréhez egy x nevű segédváltozót használunk.

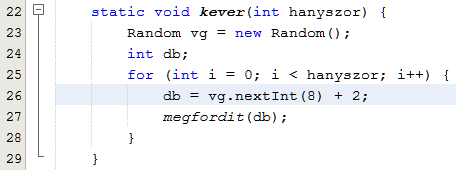
Alakítsd át ilyenre a programot, majd próbáld ki többféle n értékkel is! Az alábbi mintán az első 5 elemet fordítottuk meg.



## Keverés

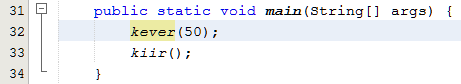
A keverést vissza lehet vezetni a fordításra: Ha egymás után többször megfordítjuk a lista elejének véletlenül kiválasztott hosszúságú részét, a lista összekeveredik. Elméletileg előfordulhat, hogy éppen az eredeti állapot lesz az eredmény, de kellő számú fordítás esetén ennek esélye igen kicsi.

Ebben a változatban készítünk egy kever() metódust, és teszteljük is. A program elején importáld a java.util.Random csomagot, majd írd be a kever() metódust a fordit() metódus után (lehetne elé is, a metódusok sorrendjének nincs jelentősége):



Figyeld meg a véletlen szám előállítását a 25. sorban! Mekkora véletlen számok lehetnek? Írd ide:

A main() metódust alakítsd át ilyenre:



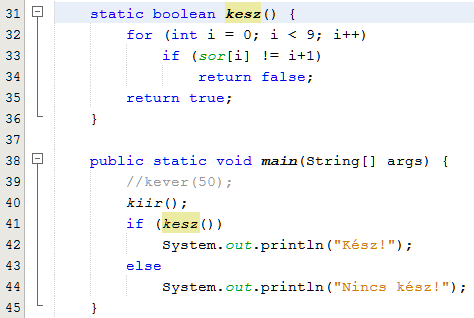
Próbáld ki! Ahányszor futtatod, mindig más eredményt kapsz.



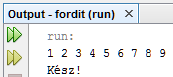
## Kész-e a rendezés?

Szükségünk lesz olyan metódusra is, amely eldönti, hogy sorban vannak‑e a számok. Ehhez egy ciklussal megyünk sorban a számokon. Ha valamelyik nem egyezik meg a ciklusváltozó+1-gyel, akkor nem jó a sorrend. Ha mind megegyezik, akkor sorban vannak a számok.

Írd be a kesz() metódust, és módosítsd a főprogramot így:



Próbáld ki így is (eredmény lent) és úgy is, hogy a 39. sorban törlöd a //-t. Akkor összekeveri a számokat a vizsgálat előtt.

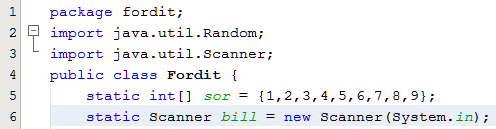


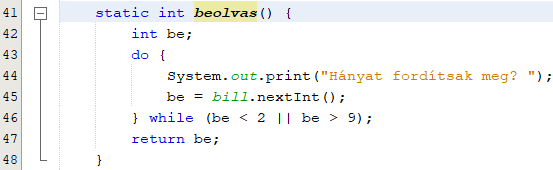
## Beolvasás

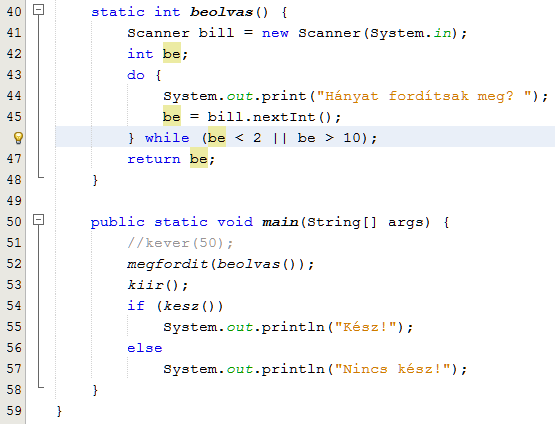
Még egy metódust fogunk használni a programban, amellyel beolvassuk, hogy a játékos hány számot szeretne megfordítani.

A beolvasáshoz importálnunk kell a Scanner osztályt. Mivel ezt a beolvas() metódusban és a főprogramban is használni fogjuk, a Fordit osztály elején létrehozunk belőle egy példányt.

Alakítsd át a program elejét így:

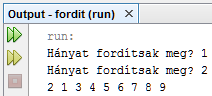


A beolvasást ellenőrizni kell, csak 2 és 9 közötti számokat lehet elfogadni. Írd be a beolvas() metódust a main() metódus elé, és alakítsd át a főprogramot így:  




Figyeld meg az 52. sorban, hogy a beolvasást eredményét közvetlenül átadjuk a megfordit() metódusnak! Így nem kell azt változóban tárolnunk.

Próbáld ki! Figyeld meg, hogy rossz érték beírása után újra kéri azt!



## Főprogram

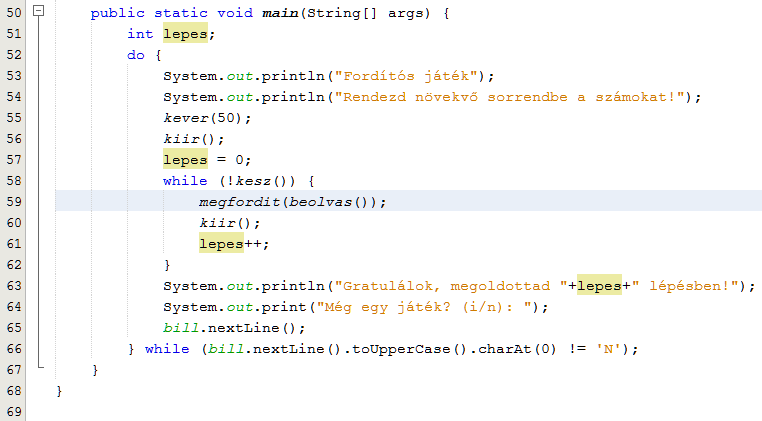
Elkészült minden szükséges metódus, már csak a main() metódust kell átalakítanunk a játékhoz.

Gondoljuk végig, mit kell ennek csinálnia egy játék alatt!

1. Kiírja, hogy mi ez, összekeveri a számokat, kiírja őket, és a lépésszámot 0-ra állítja.
2. Amíg nincsenek sorban a számok, megfordít annyit, amennyit beolvasott, újra kiírja őket és megnöveli a lépésszámot.
3. Beolvassa, hogy szeretne-e a játékos még egyszer játszani.

Ezt a három lépést addig kell ismételni, amíg a játékos nem kis- vagy nagy N-nel válaszol a 3-dik kérdésre.

A megoldás:



Készítsd el és próbáld ki!

## Stratégia

Milyen stratégiával lehet megoldani a feladatot? Mielőtt tovább olvasnál, görgess vissza az szöveg elejére, és nézd meg, hogyan csináltuk! Először próbáld meg önállóan kitalálni a módszert!

A megoldás nagyon egyszerű: Először a lista utolsó elemét tesszük a helyére, majd az utolsó előttit és így tovább a másodikig (az első ekkor már a helyén lesz).

Egy elem helyre viteléhez először annyit fordítunk, hogy az elem a lista elejére kerüljön. Ahhoz, hogy a lista elejéről a helyére tegyük a számot, pont annyit kell forgatni, amennyi a szám értéke (pl. 5-ös esetén 5-öt). Így már biztosan el tudjuk végezni a rendezést, de célszerű közben figyelni a rövidítési lehetőségeket.

Rendezd a számokat először csak gépiesen, a fenti módszerrel! A következő játékban már törekedj arra, hogy minél kevesebb lépéssel oldd meg a feladatot!

# 04. Tömbök 4.

## Hőmérséklet



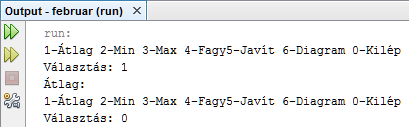
Ebben és a következő leckében egy adott év februári hőmérsékleti adatait tartalmazó tömbbel fogunk dolgozni. A tömbben egy adott helyen az aznapi átlaghőmérséklet van, egészekre kerekítve. A 0. helyen a február 1-i, az 1. helyen a február 2-i, ..., az utolsó helyen a február 28-i.

A program többféle műveletet tesz lehetővé az adatokkal. A kívánt műveletet menüből választhatjuk ki.

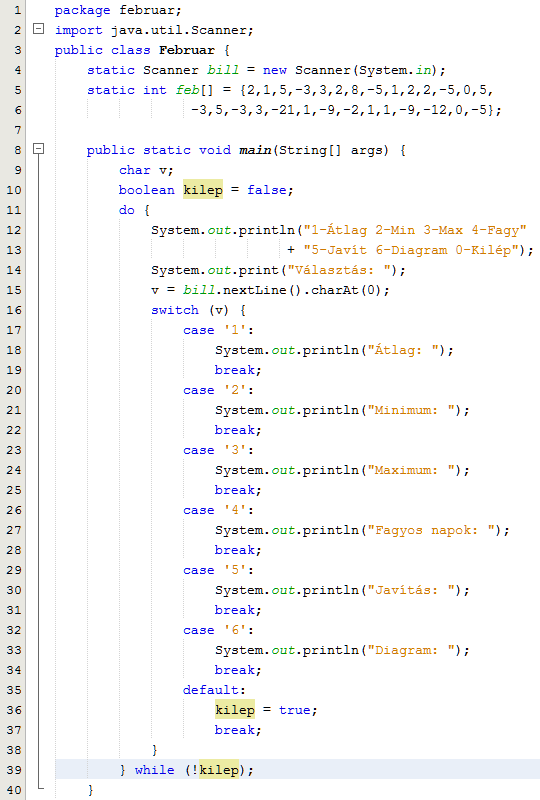
Először a program vázát fogjuk elkészíteni, és utána fogjuk kidolgozni az egyes funkciókat.

Az első változat kiírja a menüt, beolvassa a választást, majd kiírja, hogy mit kellene csinálnia. Létrehozunk egy tömböt is (feb), amely az adatokat tartalmazza.

Minta:



Készítsd el az első változatot az alábbiak szerint:



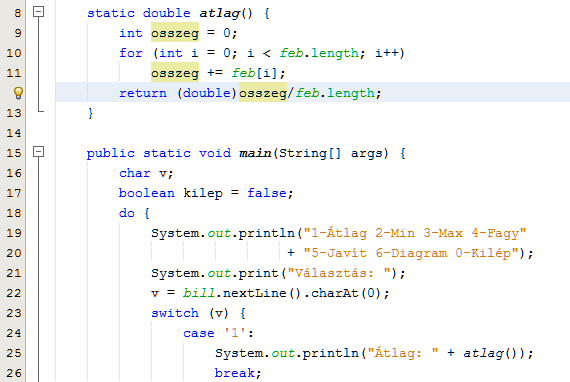
Figyeld meg, hogyan állítjuk be a kilépés feltételét!

Próbáld ki a programot többféle választással is! Mi történik, ha 7-est választanak?

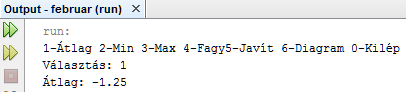
## Átlag számítása

Az 1-es menüpont kiválasztásakor a hőmérsékletek átlagát kell kiíratni. Ehhez összeadjuk a számokat, majd elosztjuk a tömb hosszával.

Az atlag() metódust a main() metódus elé írd be! A main() metódusban is módosítsd az 1-es menüponthoz tartozó részt (25. sor).

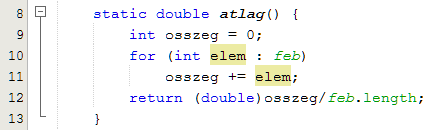


Módosítsd a programot, és próbáld ki! (A program eleje és a vége nem változik.)



## Egyszerűbb for ciklus

Ha egy tömb minden egyes elemét egymás után fel kell használni, akkor egyszerűbb formában is írhatjuk a ciklust:



A for zárójelében deklarálunk egy változót (elem), amely sorban egymás után felveszi a feb tömb elemeinek értékét, és minddel végrehajtja a ciklust.

Az egyszerűsített for ciklust (vagy más szóval for-each ciklust) csak akkor alkalmazhatjuk, ha:

* a tömb minden elemével végre kell hajtani a ciklus utasításait,
* a ciklusban nincs szükségünk az i értékére, és
* nem akarjuk módosítani a tömb elemeinek értékét.

Alakítsd át így a programot, majd próbáld ki!

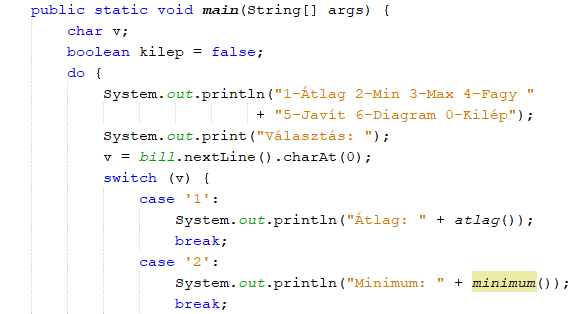
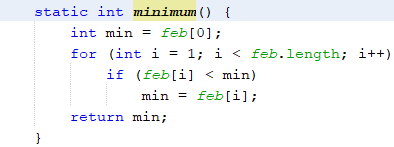
## Minimum számítása

A minimum kiszámítása a következő módon történhet:

1. Először a tömb 0-dik eleme lesz a minimum.
2. Utána sorban végigvesszük a tömb elemeit, és ha valamelyik kisebb, mint az eddigi minimum, akkor onnan kezdve az lesz a minimum.

Próbáld meg fejben az itt leírt módszert, majd készítsd el a minimum metódust, és módosítsd a főprogramot! (A program eleje és vége most sem változik.)

Most nem használhatjuk az egyszerűsített for ciklust, mert a tömb 0. elemére nincs szükségünk a ciklusban.



Próbáld ki!



## Feladatok

Az előzőek alapján készítsd el önállóan a maximumszámítást, és próbáld is ki!

Mennyi volt a legmagasabb hőmérséklet? Írd ide:

# 05. Tömbök 5.

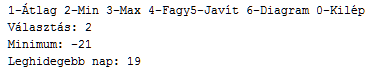
## Előkészület



A mai leckében folytatjuk az előző órán elkezdett feladatot. Nyisd meg a *februar* projektet!

## Melyik napon volt a leghidegebb?

Az előző leckében már megkerestettük a legkisebb értéket a tömbben, de nem határoztuk meg, hogy az melyik napon volt. Írassuk ki ezt is! Minta:

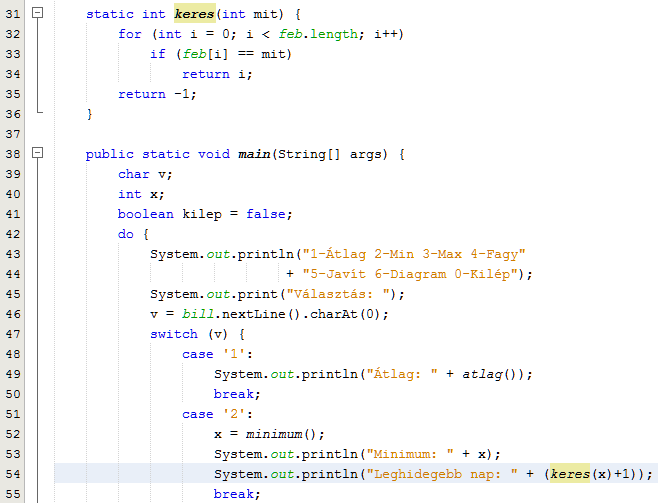


Hogyan keresünk meg egy értéket a tömbben?

Elkezdjük sorban nézni az elemeket. Ha valamelyik egyenlő a keresettel, akkor annak az indexe az eredmény. Ha a tömb végére értünk és nincs meg a keresett elem, akkor nem szerepel a tömbben.

Ha többször is szerepel egy érték a tömbben, akkor az első előfordulását fogjuk így megtalálni.

Készítsd el ennek megfelelően a keres() metódust, és módosítsd a főprogramban a 2-es ágat az alábbiak szerint:

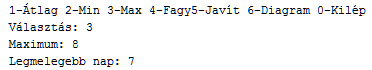


A megtalált elem indexéhez hozzá kell adni egyet, mert a tömbben 0-val kezdődik a számozás, a naptárban meg 1-gyel. Az összeadást (54. sor) még ráadásul zárójelbe is kell tenni, hogy jó legyen az eredmény. Mit írna ki a zárójel nélkül? Miért?

Próbáld ki!

(Most nem használhattuk az egyszerűsített for ciklust, mert a ciklusváltozó értékét kellett meghatározni.)

Módosítsd a programot az előzőek mintájára úgy, hogy a legmelegebb nap dátumát is írja ki! Minta:



## Fagyos napok száma

A 4. pont választásakor meg kell számolnunk, hogy hány napon volt 0 fok vagy az alatti hőmérséklet.

Készítsd el a fagy() metódust az egyszerűsített for ciklussal:



Módosítsd a főprogramot:



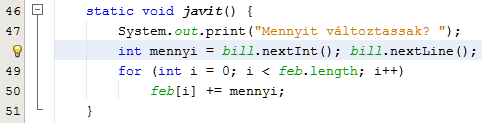
Próbáld ki!

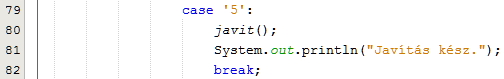


## Javítás

Hitelesítették a hőmérőt, és kiderült, hogy az egész hónap folyamán 1 fokkal kevesebbet mutatott a tényleges értéknél. Készítsünk metódust, amellyel az ilyen hibákat lehet javítani! A kapott értéket adja hozzá a lista minden eleméhez! Hogyan lehetne csökkenteni az értékeket ezzel a metódussal?

Készítsd el a javit() metódust, és módosítsd a főprogramot!





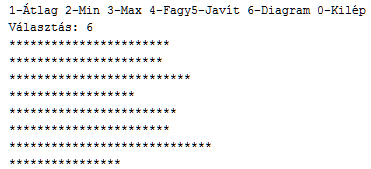
Próbáld ki 1-gyel! Az eredményt most nem fogod látni.

Miért nem használhattuk az egyszerűsített for ciklust?

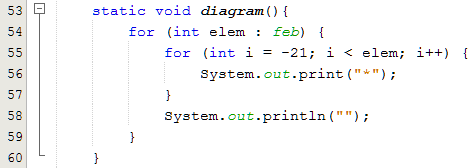
## Diagram

A diagramunk csillagokból fog állni. Minden nap sorába annyi csillag kerül, ahány fokkal melegebb volt aznap -21 foknál. (Az volt a leghidegebb.)

Minta:

  
…

A diagram() metódus:



A főprogramban csak ennyit módosíts:



Próbáld ki!

## Feladat

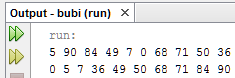
Próbáld ki még egyszer a program minden funkcióját!  
Nézd végig a program kódját, és győződj meg róla, hogy mindent értesz!

# 06. Tömbök 6.

## Feladat

Készítsünk programot, amely létrehoz egy 10 elemű, 0 és 99 közötti véletlen számokat tartalmazó tömböt! Először írja ki a számokat az eredeti sorrendben, utána rendezze őket növekvő sorrendbe, végül írja ki őket úgy is!

Minta:



## Buborékos rendezés

A legegyszerűbb rendezési módszer a buborékos rendezés. Ennél először az utolsó elemet tesszük a helyére, utána az utolsó előttit, és így tovább. Legutoljára a második és vele együtt az első elem kerül a helyére.

Az utolsó elem úgy kerül a helyére, hogy az első elemtől indulva minden elemet összehasonlítunk az utána következővel, és ha szükséges, megcseréljük őket.

Az utolsó előtti elemet ugyanígy tesszük a helyére, de ennél már csak az utolsó előtti elemig kell összehasonlítani, mert az utolsó már biztosan a helyén van.

Így sorban helyére tesszük a többi elemet is.

Utoljára a második elemet úgy tesszük a helyére, hogy összehasonlítjuk az elsőt a másodikkal, és szükség esetén megcseréljük őket.

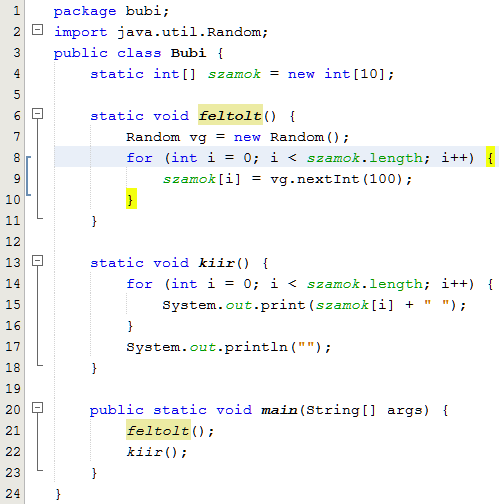
Kövesd végig a rendezést az alábbi ábrán! A színek jelentése:

* kék: összehasonlítás, nem kell csere
* piros: összehasonlítás, kell csere
* szürke: már a helyén van



## A feladat megoldása

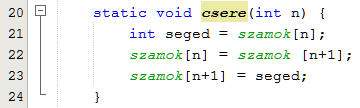
A program első változatában létrehozzuk a tömböt, feltöltjük véletlen értékekkel, majd kiíratjuk az elemeit:



Próbáld ki!



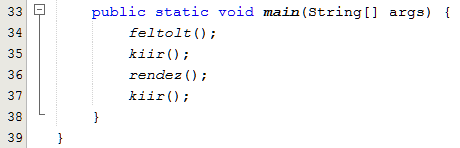
A rendezés elvégzéséhez szükség lesz egy csere() metódusra, amely megcseréli az n-dik és az n+1-dik elemeket. Írd a kiir() metódus után:



Most következik a buborékos rendezés:



És végül meg kell hívni a rendezést a főprogramból:



Próbáld ki a teljes programot!

## Rendezés csökkenő sorrendben

A csökkenő sorrendben történő rendezéshez csak egy apró módosításra van szükség. Végezd el ezt, és próbáld ki!

## Stringek rendezése

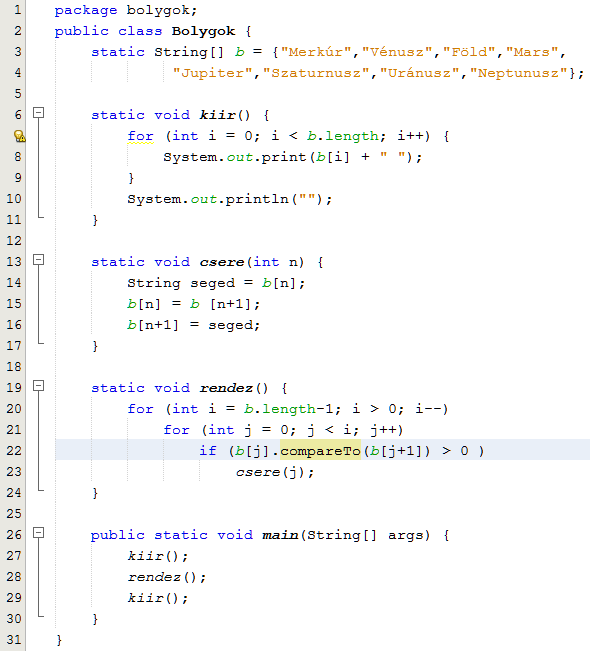
Stringeket nem hasonlíthatunk össze a kisebb és nagyobb jelekkel. A stringek összehasonlításához a compareTo() metódust használhatjuk. Például:

String a = "alfa";  
String b = "béta";  
if (a.compareTo(b) < 0) System.out.println("a kisebb, mint b");

A compareTo() metódus a pont előtti stringet hasonlítja össze a zárójelben lévővel (a-t b-vel). Ha az első előrébb van a rendezésnél, akkor az eredmény negatív, ha hátrább, akkor pozitív. Ha a két string megegyezik, akkor az eredmény 0.

A fenti példában az if utasítás feltétele igaz lesz.

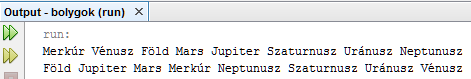
A stringek rendezését a *bolygok* nevű projekttel fogjuk kipróbálni. Ezt már előkészítettük, csak nyisd meg, és hasonlítsd össze a *bubi* programmal!



Nézzük a különbségeket:

* Most nem számokból, hanem stringekből álló tömböt rendezünk, amelyet néhány bolygó nevével töltünk fel.
* A rendezésnél nem > jellel, hanem a compareTo() metódussal hasonlítjuk össze a tömb két elemét.

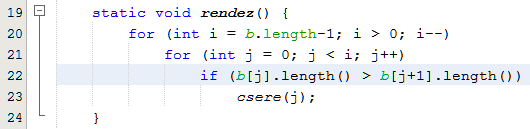
Próbáld ki a programot!



## Rendezés más szempont szerint

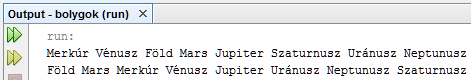
Előfordul, hogy nem ABC rendbe, hanem valamilyen más szempont alapján szeretnénk rendezni. Legyen most az a feladat, hogy a bolygók nevének hossza szerint rendezzünk! A legrövidebb nevű kerüljön előre, a leghosszabb nevű pedig a végére!

Ehhez csak a 22. sort kell átírni a rendez() metódusban:



Ebben a feltételben tetszőleges összehasonlítást megadhatunk, és a program e szerint fog rendezni.

Próbáld ki!



Megjegyzés: A buborékos rendezés a legegyszerűbb rendezési módszer, de egyben a legkevésbé hatékony is. Más szavakkal: sok (ezer vagy millió) adat esetén elég lassú. A mi feladatainkban ez nem probléma, de komoly programokban másik módszert kell alkalmazni.

## Feladat

A következő videó a buborékos rendezést szemlélteti:  
<https://www.youtube.com/watch?v=nY3As6NPmeQ>

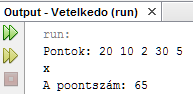
Nézd meg!

Keress, és nézz meg olyan videókat, amelyek táncokkal szemléltetik a különböző rendezési módszereket!

# 07. Tömbök 7.

## Vetélkedő

A mai feladatban beolvassuk egy vetélkedő egy játékosának pontszámait, kivesszük közülük a legkisebbet, majd a többit összeadjuk és kiírjuk. Nem tudjuk előre, hogy hány pontszám lesz, csak azt, hogy 10-nél nem több, ezért egy nem szám karakter (pl. x) beírásával jelezzük, hogy nincs több szám.



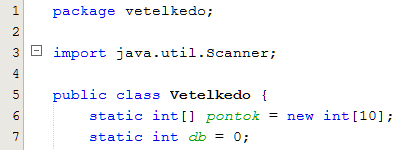
Eközben megismerkedünk az Arrays osztály néhány metódusával.

## Tömb elemeinek beolvasása

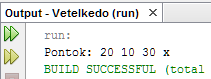
Először be kell olvasnunk a billentyűzetről egy tömb elemeit.

Egy 10 elemű tömböt deklarálunk, és számoljuk, hány elemet olvastunk be.

Kezdj egy új projektet *vetelkedo* néven! Hozz létre a Vetelkedo osztályban egy pontok nevű, 10 egész számot tartalmazó tömböt, és egy db változót, amelynek kezdőértéke legyen 0!



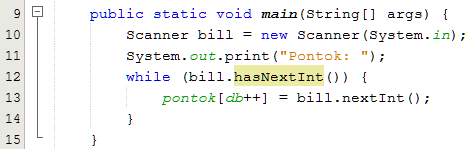
A program csak egyszer kérdezi meg a pontokat, de ezután több pontszámot is beírhatunk szóközökkel, tabulátorokkal és sorvégjelekkel (Enterekkel) elválasztva. A számsorozat végét egy olyan karakterrel jelezzük, amely nem értelmezhető számként vagy elválasztójelként. Az alábbi példában ez az x:



Ebben az esetben a pontok tömb 0. elemébe a 20, 1. elemébe a 10, 2. elemébe pedig a 30 kerül majd. A db változó értéke 3 lesz.

A beolvasáshoz szükségünk van a Scanner osztály egy újabb metódusára, a hasNextInt() metódusra. Ez jelzi, hogy a bemeneten a következő karakterek értelmezhetők-e egész számként vagy elválasztójelként. Az Enter megnyomása után vár a további karakterekre.

A beolvasást így valósítjuk meg:



Írd be, és próbáld ki különböző hosszúságú számsorozatokkal! A számokat külön sorokba is írhatod, a sorozat végét jelezd egy tetszőleges betűvel!

## Az Arrays osztály

Az Arrays osztály metódusai a tömbökkel kapcsolatos feladatok megoldását könnyítik meg. Nézd meg először a hivatalos online dokumentációt: <https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html>

Az alábbi táblázatban kiemeltünk néhány fontos metódust:

|  |  |
| --- | --- |
| metódus | leírás |
| copyOf(t,n) | Másolatot készít a t tömb első n eleméről egy új tömbbe. |
| toString(t) | Stringgé alakítja a t tömböt, amelyet ki lehet íratni. |
| sort(t) | Növekvő sorrendbe rendezi a t tömböt. |

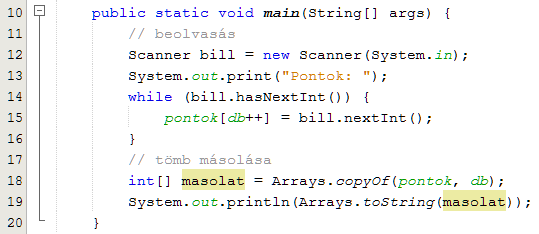
Az egyes metódusoknak többféle változata is van, hogy különböző típusú tömbökkel is lehessen használni őket.

Ezek a metódusok osztálymetódusok, vagyis híváskor az osztály nevét (Arrays) kell eléjük írni.

## Tömb másolása, kiíratása

Programunkban először másolatot készítünk a pontok tömbről, majd ezt kiíratjuk. A masolat tömb mérete akkora lesz, hogy éppen elférjenek benne a beírt pontok. Ezt úgy érjük el, hogy db számú elemet másolunk.

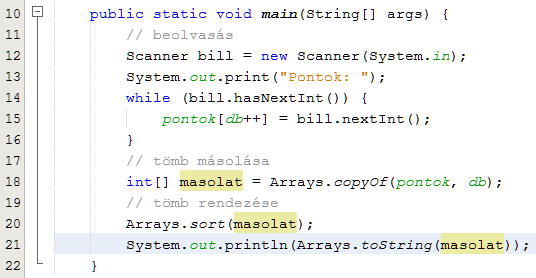
Egészítsd ki a main metódust az alábbiak szerint:



Próbáld ki! Figyeld meg a kiírás formátumát!

## Tömb rendezése

A következő lépés a masolat tömb rendezése. Egészítsd ki a main metódust így, majd próbáld ki!

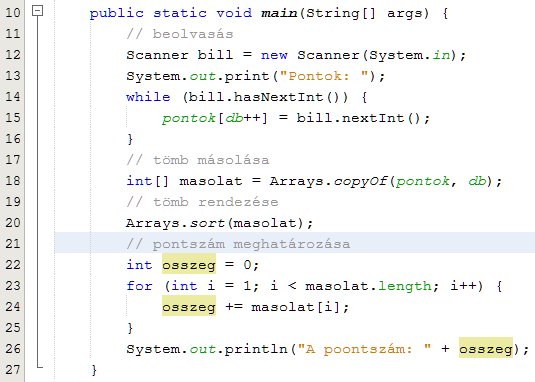


Az Arrays.sort() metódus nagyon gyors rendezési algoritmust valósít meg, ezért lehetőleg ezt használd! (Sokkal gyorsabb, mint a buborékos rendezés ☺)

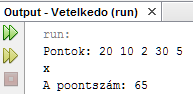
Sajnos a csökkenő sorrendbe rendezést nem lehet ilyen egyszerűen elvégezni, erről majd később lesz szó.

## Feladat befejezése

A feladat szerint a legkisebb kivételével össze kell adni a beolvasott számokat. A rendezés után ez nagyon egyszerű: a második (vagyis 1. indexű) elemtől adjuk össze a tömb elemeit:



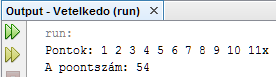
Próbáld ki!



## Önálló feladat

Egészítsd ki a programot úgy, hogy ha betelik a 10 elemű tömb, akkor már ne fogadjon további számokat! (Egyszerűen ne vegye őket figyelembe!)

Próbáld ki!



## Wrapper (csomagoló) osztályok

A Java nyelvben vannak primitív és referencia típusok. A primitív típusok egy egyszerű értéket tartalmaznak (szám, karakter vagy logikai érték), a referencia típusok pedig egy objektumnak a címét (pl. string, tömb, …)

Vannak olyan esetek, amikor csak referencia típust lehet alkalmazni, de mi egyszerű adatokat szeretnénk megadni. Például a tömbök csökkenő rendezését csak akkor tudjuk beállítani az Arrays.sort() metódusban, ha a tömb elemei referencia típusúak, vagyis objektumok.

Ilyen esetekben az egyszerű adatokat betehetjük, becsomagolhatjuk egy-egy objektumba (ahogyan az ábrán a sütiket a csomagolásba).



Ha például egész számokból szeretnénk objektumokat csinálni, akkor az Integer osztályt használhatjuk. Az átalakítás mindkét irányban teljesen automatikus (auto-boxing):

Integer n = 23;  
int x = n + 1;

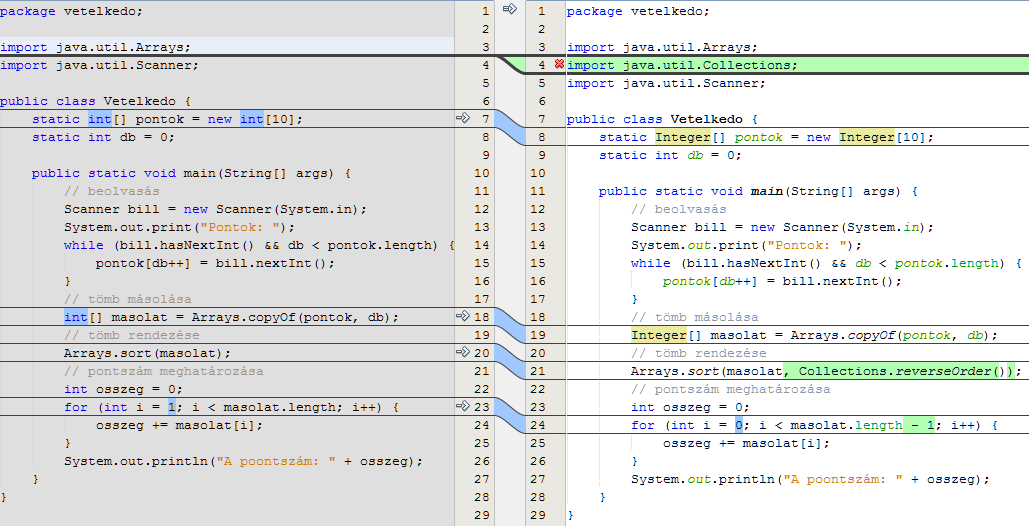
A következő táblázat mutatja, hogy milyen wrapper (csomagoló) osztályokat használhatunk:

|  |  |
| --- | --- |
| **primitív típus** | **osztály** |
| byte | Byte |
| boolean | Boolean |
| char | Character |
| double | Double |
| float | Float |
| int | Integer |
| long | Long |
| short | Short |

## A program új változata

Ez alapján elkészítettem a program egy másik változatát, amely csökkenő sorrendbe rendezi a pontokat.

Az alábbi ábrát a NetBeansben a History gombra kattintva állítottam elő. A bal oldalon az eredeti program, a jobb oldalon az új változat látható:



Figyeld meg a különbségeket:

* Az int típus helyett az Integer típust használtam a tömböknél (8., 19. sor).
* A fordított sorrendet a Collections.reverseOrder() metódussal adtam meg (21.sor), amihez importálni kellett a Collections osztályt (4. sor).
* Mivel a legkisebb pontszám most a tömb végére került, módosítani kellett a határokat a for ciklusban (24. sor).

Írd át te is a programot, és próbáld ki!

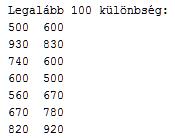
# 08. Tömbök 8.

## Feladat

Egy hegymászó az útja során többször megmérte, hogy milyen magasan jár. A mért értékek: 500, 500,500, 500, 500, 600, 600, 650, 700, 750, 820, 880, 930, 1010, 1050, 980, 930, 830, 780, 720, 720, 710, 700, 750, 770, 790, 820, 880, 880, 820, 760, 740, 600, 500, 560, 670, 780, 820, 920, 880, 860, 820, 770, 770, 760, 750, 740, 740 ,730,720.



Az alábbi feladatokat először próbáld meg egyedül megoldani, és csak utána nézd meg a megoldásokat!

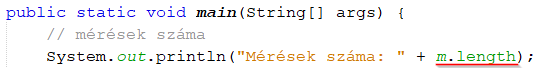
1. Nyisd meg hegyek projektet, és nézd meg a Hegyek osztály tartalmát!
2. Írasd ki a mérések számát!  
   
3. Írasd ki egymástól szóközzel elválasztva a 800 m feletti értékeket! Írasd ki azt is, hogy hány ilyen van! Minta:  
   
4. Írasd ki a legmagasabb mért értéket! Minta:  
   
5. Írasd ki azokat az egymás utáni értékeket, amelyek között legalább 100 m különbség van! Minta:  
   
6. Azokat a helyeket, amelyek előtt és után közvetlenül alacsonyabb pont van, csúcsoknak nevezzük. Közvetlenül az út elején vagy végén nem lehet csúcs. Listáztasd ki a csúcsokat az előttük és a mögöttük lévő pontok magasságával együtt! Minta:  
   

## Megoldás

### 2. feladat: Mérések száma

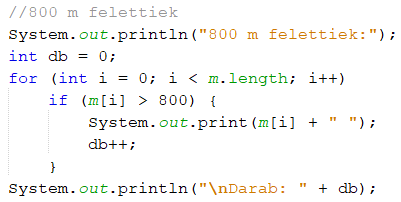
A mérések száma megegyezik a tömb méretével. Ezt a tömb length változójával lehet lekérdezni. Például: m.length

Írd ezt a main metódusba:



### 3. feladat: 800 m felettiek

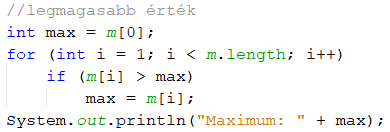
A feladatot úgy oldjuk meg, hogy sorra veszünk minden egyes tömbelemet, és ha 800 fölött van, akkor kiíratjuk, és egy számlálót megnövelünk 1-gyel.



### 4. feladat: Legmagasabb érték

A megoldásnál először feltételezzük, hogy a tömb nulladik eleme a legnagyobb. Utána végigvesszük az összes többi elemet, és ha valamelyik nagyobb, mint az eddigi maximum, akoor onnan kezdve az az érték lesz a maximum.

Folytasd így a main metódust:

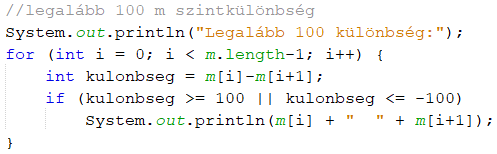


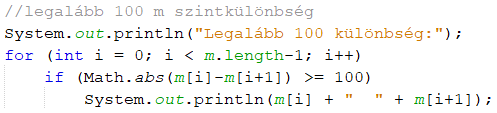
### 5. feladat: Legalább 100 m szintkülönbség

Most a ciklusban az elemeket a nulladiktól az utolsó előttiig vesszük, és minden elemet összehasonlítunk az utána következővel. Ha a különbségük nagyobb, mint 100, akkor kiírjuk őket.

A különbség kétféleképpen lehet nagyobb, mint 100. Vagy legalább 100 m emelkedés, vagy legalább 100 m lejtés van a két pont között.

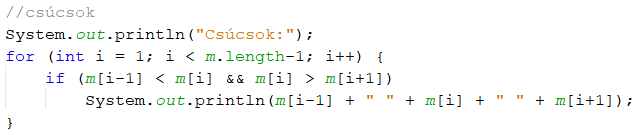
Folytasd így a main metódust:



A feltételt egyszerűsíthetjük úgy, hogy a különbséget nem tesszük külön változóba és az abszolút értékét vesszük:  


### 6. feladat: Csúcsok

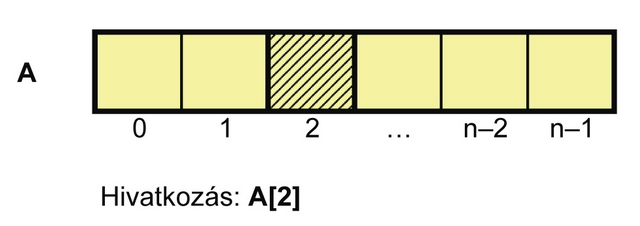
A csúcsok vizsgálatánál a 2. elemtől az utolsó előttiig megyünk, és minden elemet összehasonlítunk az előtte és az utána állóval. Ha ezek kisebbek, mint az elem, akkor a csúcson vagyunk.



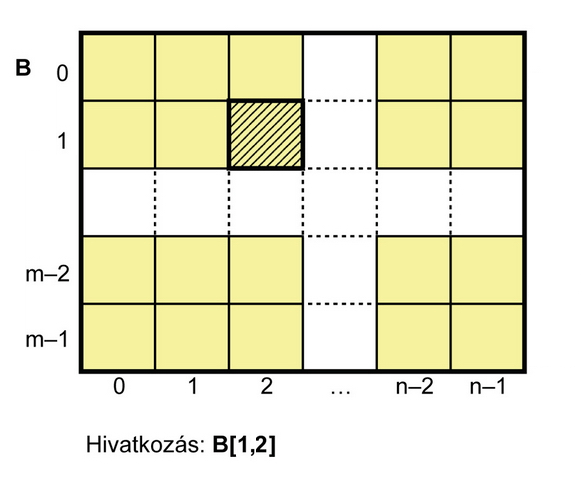
# 09. Tömbök 9.

## Kétdimenziós tömbök

Az eddigiekben egydimenziós tömbökkel dolgoztunk, amelyeket vektoroknak is szokás nevezni. Ezeket egy sorként lehet elképzelni. Tetszőleges elemük megadásához egy indexre van szükség. Például:



Használhatunk azonban kétdimenziós tömböket is, amelyeket mátrixoknak is szokás nevezni. Ezeket egy sorokból és oszlopokból álló táblázatként lehet elképzelni. Egy elemükre két indexszel lehet hivatkozni. Az egyik megadja a sorszámot, a másik pedig az oszlopszámot.

  
 Hivatkozás: **B[1][2]**

Nézzük meg, hogyan lehet létrehozni egy 2x3-as tömböt:

int[][] b = new int[2][3];

Az egész számokat tartalmazó tömbök a létrehozás után 0-kkal vannak feltöltve. Az elemeket így módosíthatjuk:

b[1][2] = 10;

Kétdimenziós tömbök létrehozásánál is fel lehet sorolni a kezdőértékeket, kapcsos zárójelek közé téve a sorokat és az egész tömböt is:

int[][] c = {   
 {1,2,3},   
 {4,5,6}   
 };

Ez az utasítás egy két sorból és három oszlopból álló tömböt hoz létre a felsorolt elemekkel.

Ha ki szeretnénk íratni egy kétdimenziós tömb összes elemét, akkor két ciklus kell: az egyik végigmegy a sorokon, a másik pedig az oszlopokon. Például:

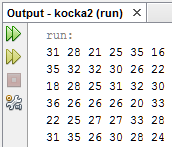
for (int i = 0; i < c.length; i++)  
 for (int j = 0; j < c[i].length; j++)  
 System.out.println(c[i][j]);

A c.length a c tömb sorainak számát adja meg, a c[i].length() pedig az i. sor elemeinek számát (vagyis az oszlopok számát).

## Két kocka



Készítsünk programot, amely ezerszer dob két kockával, majd kiírja a dobások számait! Az egyik kocka határozza meg a sort, a másik pedig az oszlopot. Minta:

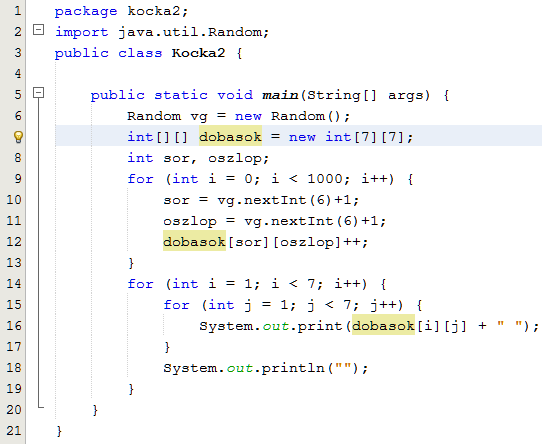


Például a 3. sor 4. eleme azt mutatja, hogy ebben a sorozatban 1000 dobásból 31-szer fordult elő az az eset, hogy az első kockával 3-ast, a másodikkal 4-est „dobott” a gép.

Hányszor „dobott” a gép dupla hatost? Írd ide:

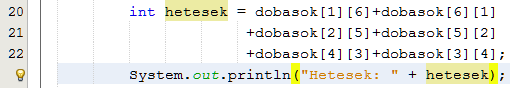
A programban valójában egy 7x7-es tömböt hozunk létre (dobasok), de a 0. sorral és a 0. oszloppal nem foglalkozunk. A kiírásnál is az 1. sorral és az 1. oszloppal kezdünk. Kihasználjuk, hogy az egész számokból álló tömbök induláskor 0-kkal vannak feltöltve.

Nézzük a programot:



Próbáld ki a programot! Módosítsd úgy, hogy tízezerszer dobjon, és próbáld ki így is!

Szeretnénk azt is kiíratni, hogy hányszor lett a dobások összege 7. Ez az összeg többféleképpen is kijöhetett (1 és 6, 6 és 1, 2 és 5, 5 és 2, 4 és 3, 3 és 4). Ezeket az eseteket össze kell adni:



Szúrd be ezeket a sorokat a main() metódus végére, majd próbáld ki!

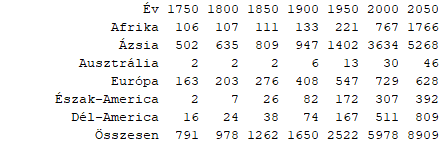
## Önálló feladat:

Írasd ki azt is, hányszor volt a dobások összege 11!

# 10. Tömbök 10.

## Feladat

Ebben a feladatban az egyes földrészek népességét kell kiíratni a megadott adatok alapján, táblázatos formában:

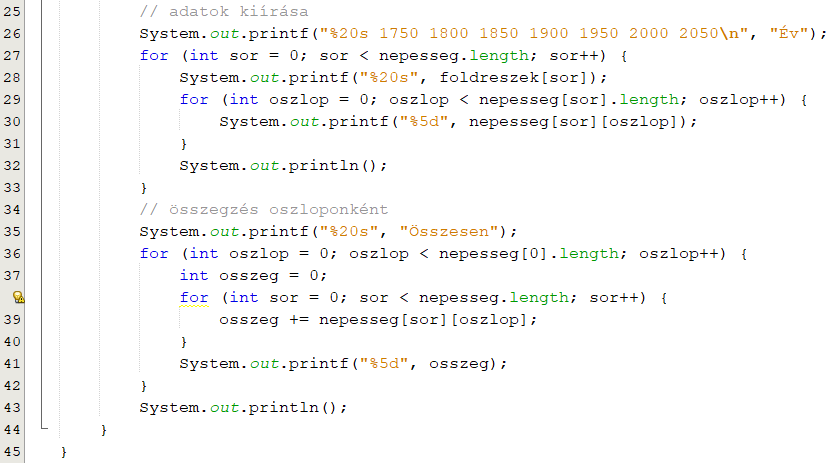


Nyisd meg a kapott *nepesseg* nevű programot, és oldd meg a következő feladatokat!

1. A tömbök inicializálását már elvégeztük. Nézd meg a kódot! A földrészek olyan sorrendben vannak, mint a sorok a kétdimenziós tömbben.
2. Írasd ki az adatokat a fenti minta alapján! A földrészek oszlop szélessége 20 karakter, az adatok pedig 5 karakter széles oszlopokban vannak.
3. Összegezd az oszlopok adatait, és írasd az eredményeket az utolsó sorban a minta szerint!

## Megoldás





# 11. Listák 1.

## Listák

A tömbök használatánál előre meg kell adni a tömb méretét, az később már nem változhat (legfeljebb átmásolhatjuk az elemeket egy másik tömbbe).

Amikor nem tudjuk előre a tárolandó elemek számát, akkor kényelmesebb egy lista (ArrayList) használata. A lista mérete tetszőlegesen változhat, bármikor hozzáadhatunk új elemet, vagy törölhetünk belőle egy elemet.

Az elemeknek azonos típusúaknak kell lenniük. Ez a típus egy tetszőleges osztály lehet. A primitív típusokat itt is be kell csomagolni (Integer, Double, …)

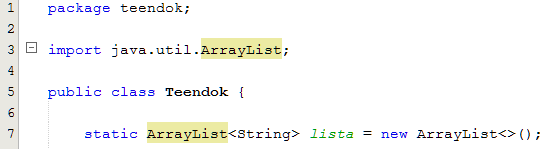


## Teendők program

Mai példánkban egy teendő listát (to-do list) fogunk készíteni.

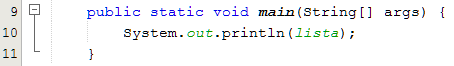
Kezdj egy programot *teendok* néven!

Hozz létre a programban egy lista nevű ArrayList<String> típusú változót! Ez egy olyan lista, amely String típusú elemeket tartalmaz majd. A használatához importálni kell az ArrayList osztályt:

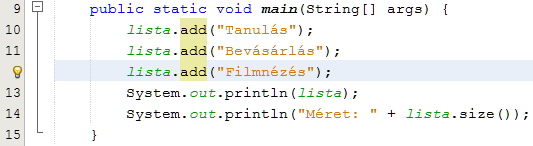


A <> jelek között kell megadni az egyes elemek típusát. A jobb oldalon is megadhattuk volna a <String> típust, de ez nem kötelező.

A főprogramban írasd ki az üres listát!

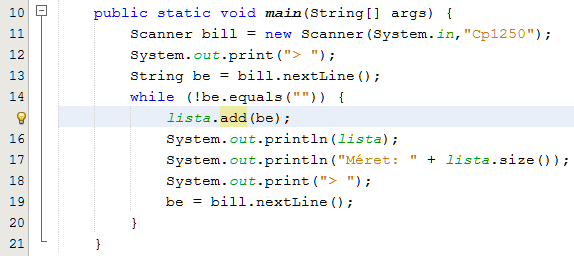


Adj hozzá néhány teendőt a listához! A hozzáadás az add() metódussal történik, a lista méretét pedig a size() metódussal lehet lekérdezni:



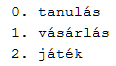
## Beolvasás

Természetesen nem előre megadott tevékenységeket szeretnénk a listába, hanem be szeretnénk olvastatni az elemeket, ezért alakítsd át így a programot:

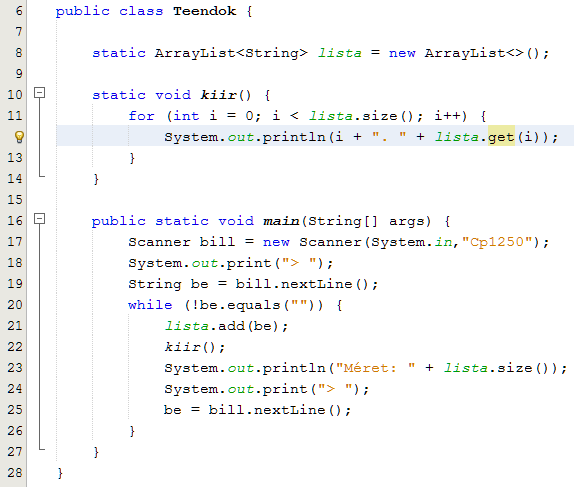


Próbáld ki! Sajnos minden alkalommal újra kell írni a teendőket, mert még nem tudjuk fájlba menteni őket. Nemsokára megtanuljuk majd azt is.

Szeretnénk javítani a kiíráson úgy, hogy számozva, egymás alá írja ki a teendőket. Például:



Ehhez készítünk egy kiir() metódust, és ezt hívjuk meg a kiíráshoz. Ebben a get() metódust használjuk a lista elemeinek lekérdezéséhez. A lista.get(i) a lista i-dik elemét adja meg.

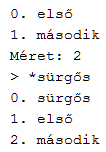


Próbáld ki!

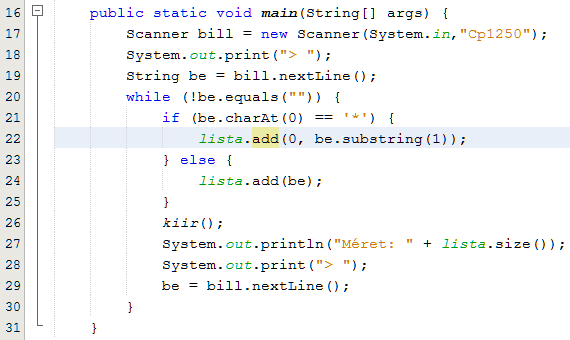
Ha a ciklusban nem lenne szükségünk az i értékére, akkor itt is alkalmazhatnánk az egyszerűbb for-each ciklust, mint a tömböknél.

Sürgős teendőinket nem a lista végéhez szeretnénk fűzni, hanem az elejére szeretnénk beszúrni. Ebben az esetben az add() metódusnak azt a változatát használjuk, amelyben azt is meg lehet adni, melyik elem elé szeretnénk beszúrni. Például a lista.add(0,”sürgős”) a lista 0-dik eleme elé szúrja be a sürgős elemet.

A teendő beírásakor az elejére írt csillaggal jelezzük, hogy a teendő sürgős:



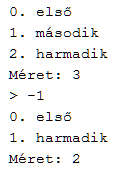
Alakítsd át így a main() metódust:



Próbáld ki!

## Törlés

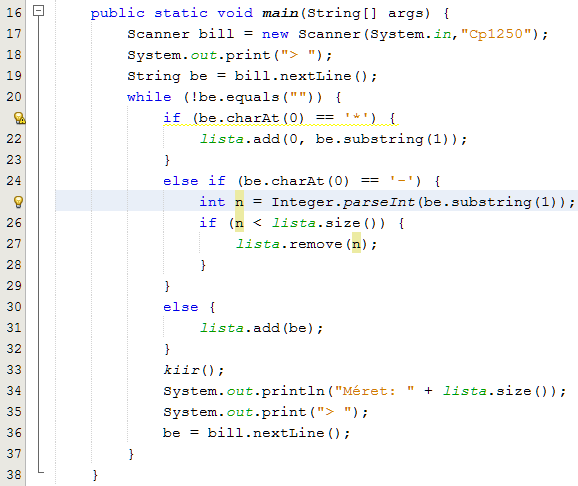
Ha valamit elvégeztünk, szeretnénk törölni a listából. Ezt egy – jellel és a teendő sorszámával fogjuk megadni. Például:



Figyeld meg, hogy a – jel után közvetlenül írtuk a számot, nem hagytunk ki helyet közöttük.

A törléshez a lista.remove(n) metódust alkalmazzuk, amely a lista n-dik elemét törli.

Ezen kívül a beolvasott string – jel utáni részét át kell alakítani egy egész számmá, mert ez adja meg, hogy hányadik elemet kell törölni. Az átalakítást elvégezhetnénk egy új Scanner objektummal, de most kényelmesebb az Integer.parseInt() metódus használata:



Próbáld ki!

## Feladat

A program nem tájékoztatja a felhasználókat arról, hogyan kell használni. Egészítsd ki úgy, hogy induláskor kiírja a legfontosabb tudnivalókat!

## Összefoglalás

Az alábbi táblázat összefoglalja a legfontosabb listákkal kapcsolatos metódusokat. Melyiket nem használtuk a programban?

|  |  |
| --- | --- |
| **Példa** | **Leírás** |
| lista.add("Mosogatás") | Hozzáadja a Mosogatás elemet a lista végéhez. |
| lista.add(1,"Vasalás") | Beszúrja az 1-es indexű elem elé a Vasalás elemet. |
| lista.set(1,"Főzés") | Átírja az 1-es indexű elemet Főzésre. |
| lista.get(1) | Megadja a lista 1-es indexű elemét. |
| lista.remove(2); | Törli a 2-es indexű elemet. |
| lista.size() | Megadja a lista elemeinek számát. |

## Feladat

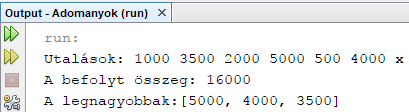
Egészítsd ki az alábbi táblázatot!

|  |  |
| --- | --- |
| **Adattípus** | **Elemek száma** |
| Array (tömb) | a.length |
| Arraylist | a. |
| String | a. |

# 12. Listák 2.

## Adományok

A mai feladatban beolvasunk egy jótékony célra érkező felajánlásokat, kiszámítjuk az összegüket, majd kiíratjuk a három legnagyobbat:



Eközben gyakoroljuk a listák használatát.



## Lista elemeinek beolvasása

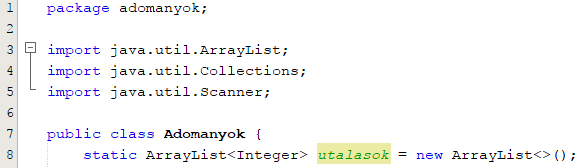
Először be kell olvasnunk a billentyűzetről a lista elemeit.

Nem tudjuk előre, hogy hány számot fognak beírni, ezért egy egész számokból álló listát fogunk alkalmazni.

Mivel a lista elemei csak objektumok lehetnek, ezért az egész számokat be kell csomagolni az Integer osztály példányaiba.



Kezdj egy új projektet *adomanyok* néven! Hozz létre az Adomanyok osztályban egy utalasok nevű, Integer típusú elemekből álló listát!



A program csak egyszer kérdezi meg az utalásokat, de a kérdés után akárhány utalást beírhatunk szóközökkel, tabulátorokkal és sorvégjelekkel (Enterekkel) elválasztva. A számsorozat végét egy olyan karakterrel jelezzük, amely nem értelmezhető számként vagy elválasztójelként. A fenti példában ez az x.

A beolvasáshoz felhasználjuk a Scanner osztály hasNextInt() metódusát, amely jelzi, hogy a bemeneten a következő karakterek értelmezhetők-e egész számként vagy elválasztójelként. Az Enter megnyomása után vár a további karakterekre.

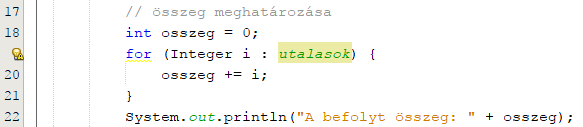
Ahogyan az előző leckében tanultuk, a listához az add() metódussal lehet újabb elemet hozzáfűzni.

A beolvasást így valósítjuk meg:



## Az összeg meghatározása

Az utalások összegének meghatározását egy egyszerűsített for ciklussal végezzük el, amely a lista összes elemét hozzáadja az osszeg változóhoz:



Írd be, majd próbáld ki a programot!

## Listák rendezése

A listák rendezéséhez a Collections osztály sort() metódusa használható. A fordított sorrendet a Collections.reverseOrder() metódussal adhatjuk meg (mint a tömböknél). Ezért importáltuk a program elején a Collections osztályt.

Rendezzük csökkenő sorrendbe az utalasok listát:



## Listák másolása, részlisták

A listák nem másolhatók úgy, hogy egy változót egyenlővé teszünk a lista nevével. Ha például ezt írnánk:   
masolat = utalasok;  
akkor mindkét változó ugyanarra a listára mutatna.

Ehelyett létre kell hozni egy új listát, és az inicializálásnál a zárójelek között meg kell adni az eredeti listát:  
masolat = new ArrayList<>(utalasok);

Most azonban nem a teljes listát szeretnénk egy új listába másolni, csak az első három elemét. Ilyen részlistát a subList() metódussal tudunk előállítani. Például:  
utalasok.subList(0, 3)

Ez a kifejezés egy olyan részlistát állít elő, amely az utalasok lista 0. elemével kezdődik, és a 3. elem már nincs benne. Vagyis a 0., az 1. és a 2. elem lesz benne. (Ez az első három elem.)

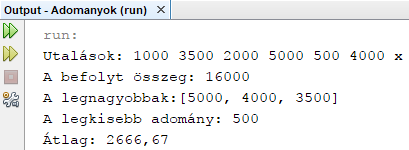
Ennek alapján így tudjuk kimásolni az utalások lista első három elemét a másolat listába:



Utána ki is íratjuk őket. Próbáld ki!

## Önálló feladatok

1. Egészítsd ki a programot úgy, hogy írja ki a legkisebb adományt is!
2. Egészítsd ki a programot még azzal is, hogy ha 0-nál több adomány érkezett, akkor írja ki az átlagukat is, két tizedesjegyre kerekítve!



## Segítség a megoldáshoz

Az alábbi táblázatban kiegészítettük a múltkori összefoglalót a legfontosabb listákkal kapcsolatos műveletekről:

|  |  |
| --- | --- |
| **Példa** | **Leírás** |
| lista.add("Mosogatás") | Hozzáadja a Mosogatás elemet a lista végéhez. |
| lista.add(1,"Vasalás") | Beszúrja az 1-es indexű elem elé a Vasalás elemet. |
| lista.set(1,"Főzés") | Átírja az 1-es indexű elemet Főzésre. |
| lista.get(1) | Megadja a lista 1-es indexű elemét. |
| lista.remove(2) | Törli a 2-es indexű elemet. |
| lista.size() | Megadja a lista elemeinek számát. |
| lista.subList(1,4) | Részlista az 1-től a 3-as indexű elemig (a 4-es indexű már nincs benne!) |

# 13. Fájlok 1.

## Bevezetés

A mai leckében egy szöveges fájlt szeretnénk beolvasni, és tartalmát kiíratni a képernyőre. Mivel azonban a fájlok használata során hibák történhetnek (pl. nem létező fájlt próbálunk megnyitni), először a hibakezeléssel kell megismerkednünk.

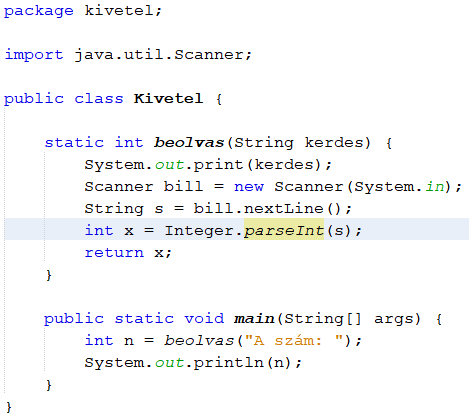
## Hibák a programban

Egy program lehetséges hibáit két csoportba sorolhatjuk:

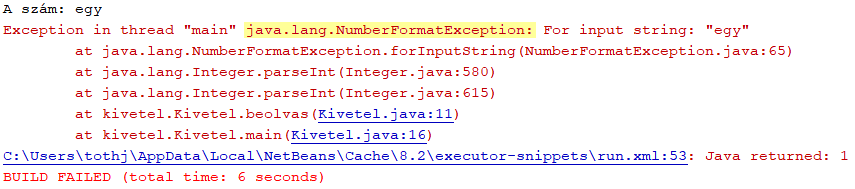
* Szintaktikai hibák: ezeket piros hullámvonallal jelzi a NetBeans, és el sem indul a program, amíg ki nem javítjuk őket.
* Futási hibák: a program szintaktikailag helyes, elindul, de futás közben valamelyik művelet eredménye hiba lesz.

Futási hibák esetén **kivétel** (**exception**) keletkezik a programban és a program hibaüzenettel leáll.

Próbaképpen készítsd el az alábbi programot:



Próbáld ki úgy hogy egy számot írsz be, és úgy, hogy szöveget! Mit tapasztalsz?



A kivételt a felső ábrán kiemelt sorban az Integer.parseInt metódus okozhatja, ha nem tudja számmá alakítani a beolvasott stringet.

Minden kivétel egy objektum, amelyet egy osztály ír le. Az alsó ábrán kiemeltük ennek a nevét (NumberFormatException).

Kétféle kivétel van. Az egyik, amit nem kötelező kezelnünk, mint a fenti példában. Ilyen esetekben a program jó megírásával elkerülhető a hiba. Például megvizsgáljuk a string tartalmát, mielőtt megpróbáljuk átalakítani. (Unchecked exception)

A kivételek másik fajtája nem kerülhető el így. Például ilyen, amikor a felhasználó rossz fájlnevet ad meg, és megpróbáljuk megnyitni. Ezeket a kivételeket kötelező kezelni a programban. (Checked exception)

Ha kivétel keletkezhet, akkor minden metódusnak kötelező azt kezelni vagy továbbadni:

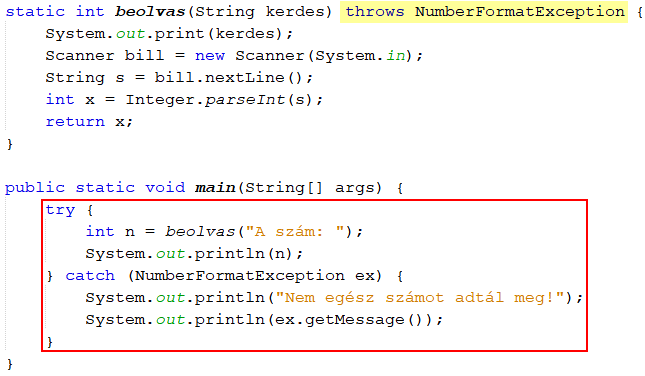
* Vagy a metódus (amelyben a kivétel történhet) fejlécében, a paraméterek után jelezzük, hogy milyen kivétel fordulhat elő. Ekkor a metódus továbbadja a kivételt az őt hívó metódusnak (vagy a main metódus esetén a Java virtuális gépnek).
* Vagy a metódusban kezeljük a kivételt. Ez úgy történik, hogy azokat az utasításokat, amelyek kivételt okozhatnak, egy try utasításba tesszük, és utána egy catch ágban megadjuk, hogy mi történjen hiba esetén. (Egy elágazásra hasonlít.)

Bár ezt a hibát nem kötelező kezelni, mi megnézzük, hogyan lehet.

Először el kell dönteni, hogy hol szeretnénk kezelni a kivételt:

* Az egyik lehetőség, hogy a beolvas metódusban kezeljük a kivételt egy try-catch szerkezettel.
* A másik lehetőség, hogy a beolvas metódus csak továbbadja a kivételt a main metódusnak, és ott végezzük el a hibakezelést. Mi most ezt fogjuk tenni.
* A main metódus fejlécében is továbbadhatnánk a kivételt. Hiba esetén ugyanúgy leállna a program, mintha nem lenne hibakezelés.

Módosítsd a programot az alábbiak szerint:

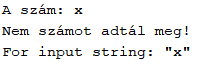


A beolvas metódusban a sárgával kiemelt rész jelzi, hogy NumberFormatException kivétel keletkezhet, és ezt a metódus továbbadja az őt hívó metódusnak. Ez most a main metódus. Ez vagy kezeli a kapott kivételt, vagy szintén továbbadhatja.

A main metódusban a try-catch szerkezet segítslglvel kezeljük a kivételt. A program megpróbálja végrehejatani a try ágban lévő parancsokat. Ha nem sikerül, akkor átugrik a catch ágba, és az ott lévő parancsokkal folytatja. a kivételeknek általában van egy getMessage nevű metódusuk, amellyel a kiveteéhez tartozó hibaüzenetet lehet megkapni.

Többféle kivételt is megadhatnánk egymás utáni catch ágakkal, és akkor a megfelelő kivételre ugrana a program, de erre most nincs szükség.

Próbáld ki a programot jó és rossz bemenő adattal is!



## Szöveges fáj beolvasása

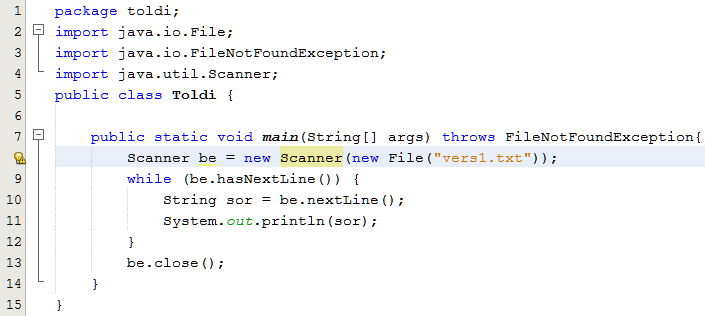
A *vers1.txt* fájl a Toldi első két versszakát tartalmazza UTF-8 kódolással, a *vers2.txt* pedig ugyanezt Windows 1250-es kódolással. A projekt létrehozása után, a program futtatása előtt másold ezeket a projekt mappájába!

Ha a projekt mappájában van a fájl, akkor nem kell megadni útvonalat a fájl neve előtt. Ha máshol van, akkor meg kell adni, ilyen formában: "T:/vers1.txt"

Egy fájl megnyitásakor a FileNotFoundException típusú kivétel következhet be, ha a program nem találja a fájlt. Ezt a kivételt kötelező kezelni. Ennél általánosabb az IOException kivétel. Ezt mindne fájlműveletnél használhatod.

Készítsünk egy olyan programot, amely megnyitja a *vers1.txt* fájlt, majd soronként beolvassa, és kiírja a képernyőre! A végén zárjuk be a fájlt!

A program első változata:



A main() metódus fejlécében (7. sor) jelezzük a kivételt, mert ez kötelezően kezelendő, és nem írtunk try utasítást. Enélkül nem lehet lefordítani a programot. Ráadásul importálni kell a megfelelő osztályokat (2-3.sor), de ehhez csak a Ctrl+Shift+I billentyűket kell megnyomnunk.

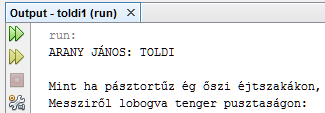
A szöveg beolvasásához egy be nevű Scanner típusú objektumot fogunk használni. A Scanner nem csak a billentyűzetről, hanem fájlból is tud olvasni. A létrehozásakor azonban (a System.in helyett) egy File típusú objektumot kell megadnunk a 8. sorban látható módon.

A Scanner objektum hasNextLine() metódusa azt adja meg, hogy van-e következő sor a fájlban. A sorokat addig olvassuk be és írjuk ki a képernyőre, amíg van következő sor.

A beolvasás után be kell zárnunk a fájlt a Scanner objektum close() metódusával.

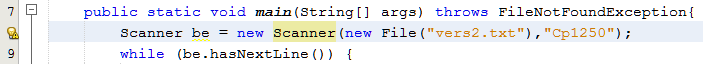
Egyébként egész vagy törtszámokat is ugyanúgy olvashatnánk be a fájlból, mint a billentyűzetről (nextInt(), nextDouble(), ...).

Másold a szöveges fájlokat a projekt mappájába, majd próbáld ki a programot!



## Kódolás megadása

Írd át a programban a fájl nevét *vers2.txt*-re, és próbáld ki így is! Azért nem jelennek meg a sorok, mert a program a kódolás miatt nem látja őket. A megoldás a kódolás megadása a 8. sorban így:



Próbáld ki így is!

## Hibakezelés

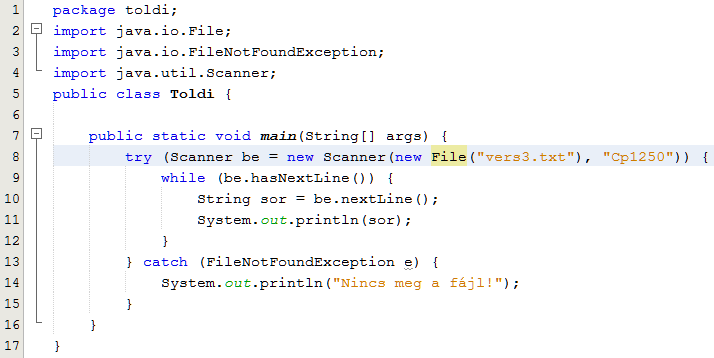
Írd át a programban a fájl nevét *vers3.txt*-re, és próbáld ki így is! Nézd meg a hibaüzenetet!



A *vers3.txt* fájl nem létezik, ezért a FileNotFoundException kivétel következik be, és leáll a program.



Nézzük meg, hogyan lehet kezelni a hibát (elkapni a kivételt):



Itt már nincs szükség a throws FileNotFoundException részre, mert a metódus kezeli a hibát.

A try után zárójelbe tesszük azokat az utasításokat, amelyek kivételt okozhatnak. Ha nem történik kivétel, akkor a try ág további részét is végrehajtja a program.

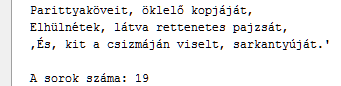
Ha kivétel történik, akkor kiválasztja a hozzá tartozó catch ágat, amelyben a megfelelő kivétel szerepel, és csak az abban szereplő utasításokat hajtja végre. (Most csak egy catch ág van, de lehetne több is.)

A végén a program automatikusan bezárja a zárójelben megnyitott fájlokat.

Próbáld ki így is, majd állítsd vissza fájlnevet *vers2.txt*-re!

## Feladat

Alakítsd úgy a programot, hogy a végén írja ki a sorok számát is! Minta:



# 14. Fájlok 2.

## Írás szöveges fájlba

A szöveg fájlba írásakor először létre kell hozni egy FileWriter típusú objektumot a fájl nevének megadásával. Például:

FileWriter f = new FileWriter("valami.txt");

Eközben az IOException nevű kivétel következhet be, amit kezelni vagy jelezni kell (throws IOException). Ez a kivétel az összes fájlműveletnél előforduló hibát magában foglalja, ezért mostantól ezt használjuk a fájlműveleteknél.

Ha a fájl még nem létezett, akkor ezzel az utasítással létrehozunk egy új szöveges fájlt, amelybe írhatunk.

Ha a fájl már létezett, akkor ezzel az utasítással töröljük a fájl korábbi tartalmát.

Ha meg szeretnénk őrizni a fájl tartalmát, és a végéhez hozzáfűzni az új szöveget, akkor a FileWriter létrehozásakor második paraméterként meg kell adni a true értéket is:

FileWriter f = new FileWriter("valami.txt", true);

Ezután létre kell hozni egy PrintWriter objektumot, amelynek paraméterként az előbb létrehozott FileWriter típusú objektumot adjuk meg:

PrintWriter p = new PrintWriter(f);

Ezután a p objektum segítségével úgy írhatunk a fájlba, mintha a képernyőre írnánk. Használhatók például a print(), a println() és a printf() metódusok. Például:

p.println("Ez egy sor.");

A program a fájlba UTF-8 kódolással ír, ezen nem is szeretnénk változtatni (bár a létrehozáskor meg lehetne adni más karakterkészletet).

A végén a p objektumot be kell zárni (p.close();). Ha ezt elmulasztjuk, előfordulhat, hogy nem kerül minden adat a fájlba.

A két lépés összevonható. Például:  
PrintWriter p = new PrintWriter(new FileWriter("valami.txt", true));

Sőt, ha nem hozzáfűzni szeretnénk, akkor a FileWriter ki is hagyható. Például:

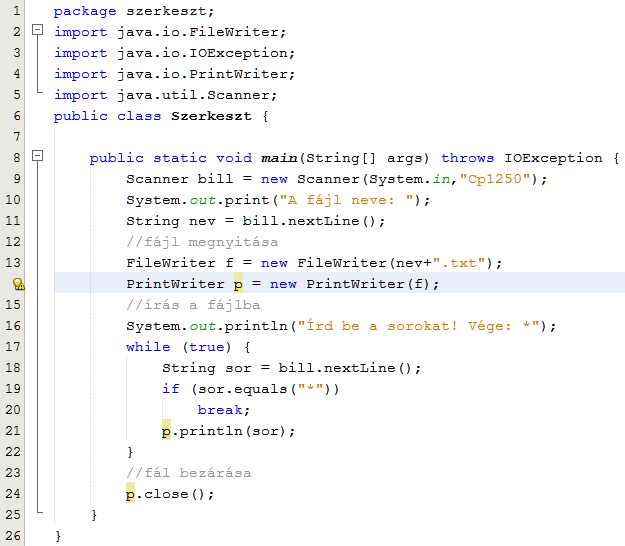
PrintWriter p = new PrintWriter("valami.txt");

## A legegyszerűbb szövegszerkesztő

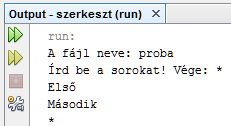
Készítsük el a legegyszerűbb szövegszerkesztőt, amely a következőket csinálja:

1. Bekéri a szerkesztendő fájl nevét, amelyhez automatikusan hozzáadja a .txt kiterjesztést.
2. Soronként beolvassa a szöveget a billentyűzetről, majd kiírja a fájlba.
3. A szöveg végét \* jelzi, ekkor bezárja a fájlt.

A program listája:

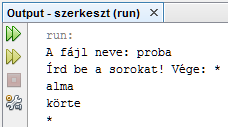


Gondold végig a program működését, majd készítsd el és próbáld ki!



Ezután ellenőrizd a *proba.txt* fájl tartalmát. A fájl a projekt mappájában jön létre.

Utána futtasd még egyszer a programot, és írd be a következőket:

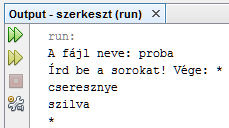


Ha most ellenőrzöd a *proba.txt* fájlt, azt tapasztalod, hogy a korábban beírt szöveg elveszett.

Alakítsd át a programot úgy, hogy mindig az előző szöveg után, a fájl végére írja az új sorokat! Ehhez csak a 13. sort kell átírni:



Próbáld ki így is:

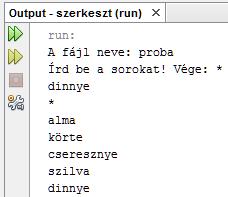


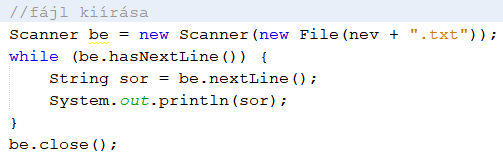
Most már a proba.txt végéhez fűzte hozzá az új sorokat. Ellenőrizd!

## Ellenőrzés

Egészítsd ki a programot úgy, hogy a \* után listázza ki a teljes fájlt!

Minta:



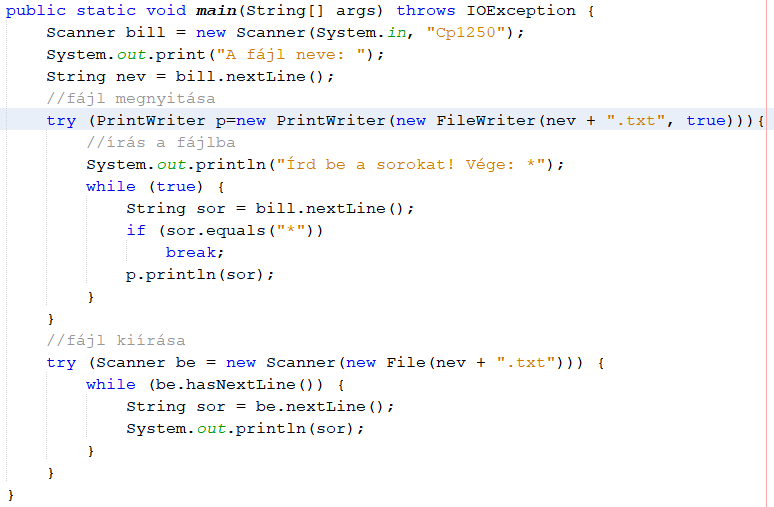


## Hibakezelés

Ezután elkészítjük a hibakezelést is:

* A kivétel továbbadása helyett try-catch szerkezetet használunk.
* A PrintWriter és a FileWriter léterehozását összevonjuk.
* A fájlokat a try után zárójelek között nyitjuk meg, ezért automatikusan bezárja őket. Így nincs szükség a close utasításokra.

Módosítsd a programot az alábbiak szerint, majd próbáld ki! Figyelj a try utáni zárójelekre (főleg a sor végén)!



# 15. Fájlok 3.

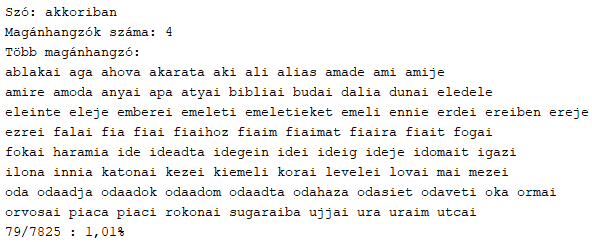
## Szavak

Ehhez a feladathoz Gárdonyi Géza Egri csillagok című regényéből gyűjtöttünk ki szavakat a *szoveg.txt* fájlba (több, mint 7 ezer szó). Az állományban csak olyan szavak szerepelnek, melyek az angol ábécé betűivel leírhatók, és minden szó csak egyszer szerepel.

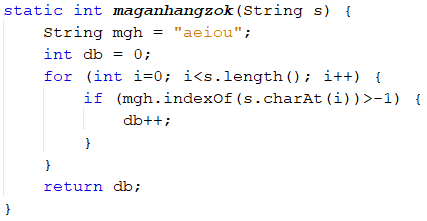
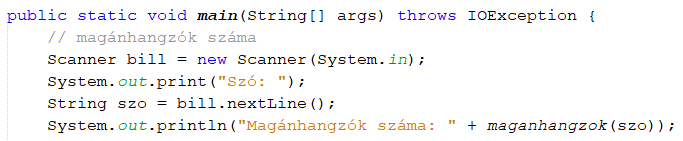
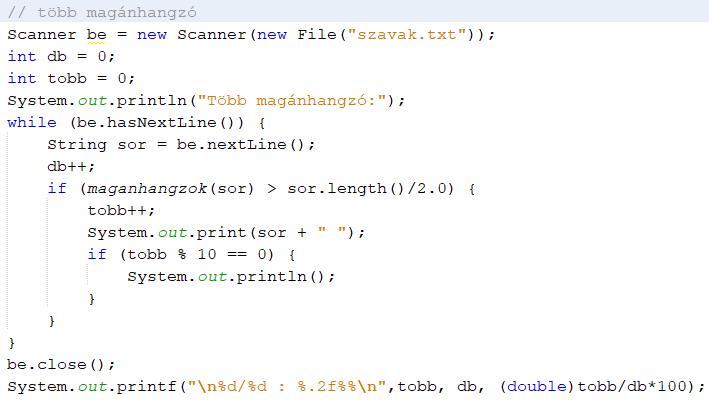
Készíts programot a következő feladatok megoldására *szavak* néven! A fájlműveleteknél a hibakeresést nem kell elkészítened.

1. Készíts egy metódust maganhangzok néven, amely megszámolja, hogy a paraméterként kapott szóban hány magánhangzó van, és ezt az értéket adja vissza!
2. Kérj be a felhasználótól egy szót, és a maganhangzok metódus segítségével írasd ki, hány magánhangzót tartalmaz! A begépelendő szóról feltételezheted, hogy csak az angol ábécé kisbetűit tartalmazza. (Az angol ábécé magánhangzói: a, e, i, o, u.)
3. A magyar nyelv szavaiban általában kevesebb a magánhangzó, mint a mássalhangzó. Határozd meg, hogy az állomány mely szavaiban van több magánhangzó, mint mássalhangzó! Ezeket a szavakat írasd ki a képernyőre egy-egy szóközzel elválasztva! Minden tizedik szó után kezdj új sort!  
   A szavak felsorolása után új sorba (a leghosszabb szó elé) a mintának megfelelően az alábbi adatokat add meg:  
    • hány szót talált;  
    • hány szó van összesen az állományban;  
    • a talált szavak hány százalékát teszik ki az összes szónak!  
   A százalék két tizedessel jelenjen meg!   
   Például: 79/7825 : 1,01%  
   Ennél a feladatnál ne olvasd be újra a sorokat fájlból, hanem az előző beolvasás feldolgozását egészítsd ki!  
   *(Segítség: a printf() metódusban a százalékjel két százalékjellel íratható ki.)*A feladatot úgy oldd meg, hogy tetszőleges hosszúságú szövegállomány esetén működjön, azaz a teljes szöveget ne tárold a memóriában!  
   Ne felejtsd el bemásolni a szavak.txt fájlt a projekt mappájába!

Minta:



## Megoldás

1. Készíts egy metódust maganhangzok néven, amely megszámolja, hogy a paraméterként kapott szóban hány magánhangzó van, és ezt az értéket adja vissza!  
   
2. Kérj be a felhasználótól egy szót, és a maganhangzok metódus segítségével írasd ki, hány magánhangzót tartalmaz! A begépelendő szóról feltételezheted, hogy csak az angol ábécé kisbetűit tartalmazza. (Az angol ábécé magánhangzói: a, e, i, o, u.)  
   
3. A magyar nyelv szavaiban általában kevesebb a magánhangzó, mint a mássalhangzó. Határozd meg, hogy az állomány mely szavaiban van több magánhangzó, mint mássalhangzó! Ezeket a szavakat írasd ki a képernyőre egy-egy szóközzel elválasztva! Minden tizedik szó után kezdj új sort!  
   A szavak felsorolása után új sorba (a leghosszabb szó elé) a mintának megfelelően az alábbi adatokat add meg:  
    • hány szót talált;  
    • hány szó van összesen az állományban;  
    • a talált szavak hány százalékát teszik ki az összes szónak!  
   A százalék két tizedessel jelenjen meg!   
   Például: 79/7825 : 1,01%  
   Ennél a feladatnál ne olvasd be újra a sorokat fájlból, hanem az előző beolvasás feldolgozását egészítsd ki!  
   *(Segítség: a printf() metódusban a százalékjel két százalékjellel íratható ki.)*A feladatot úgy oldd meg, hogy tetszőleges hosszúságú szövegállomány esetén működjön, azaz a teljes szöveget ne tárold a memóriában!  
   Ne felejtsd el bemásolni a szavak.txt fájlt a projekt mappájába!  
   

# 16. Fájlok 4.

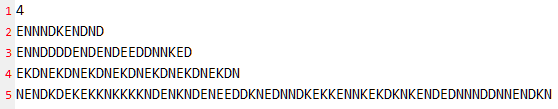
## Robot



Egy barátod robotot épített. A programozásban azonban már nem annyira jártas, ezért a segítségedet kéri.

A robot mozgását egysoros programok vezérlik. A sorokban négyféle karakter lehet: E, K, D, N. Ezek a négy égtáj (észak, kelet, dél és nyugat) irányába tett egységnyi lépéseket jelentik. Például a KKE program hatására két lépést megy keletre, majd egyet északra.

A programokat egy szöveges fájlban tárolják (*utak.txt*, UTF-8 kódolás). A fájl első sora a programok számát tartalmazza, a többi sor pedig egy-egy programot (a sorok végén sortöréssel).



## Beolvasás és kiírás

A program első változatának csak be kell olvasnia az adatokat az *utak.txt* fájlból, majd az első sor kivételével ki kell írnia egy *kimenet.txt* fájlba.

A beolvasást a beolvas() metódus, a kiírást pedig a kiir() metódus fogja végezni.

A beolvasott adatokat most a memóriában kell tárolni. Mivel itt rögtön a fájl elején beolvassuk az elemek számát, az utak beolvasása előtt létre tudjuk hozni a megfelelő méretű tömböt.

A beolvasott utakat az utak nevű, String típusú tömbben tároljuk, az utak számát pedig a db nevű egész típusú változóban. Mindkettőt használjuk majd több metódusban is, ezért a metódusok előtt, a Robot osztályon belül globális változóként deklaráljuk őket.

A hibakezelésnél az IOException kivételt adjuk meg, amely minden olvasási és írási hibához jó. A main() metódus meghívja a beolvas() és a kiir() metódusokat, és nem kezeli a kivételeket, azért a main() metódusnál is meg kell adni a kivételt.

Készítsd el a programot:



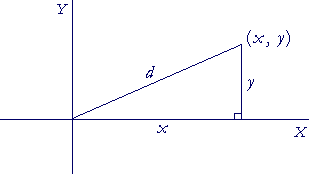
A 18. sorban a szám beolvasása után beolvassuk a sor vége jelet is, hogy az utak beolvasásánál ne okozzon hibát.

A kipróbálás előtt másold az *utak.txt* fájlt a projekt mappájába, utána pedig ellenőrizd a *kimenet.txt* fájlban az eredményt!

## Távolság

A következő változatban minden útnál kiszámítjuk, hogy az origóból indulva milyen távol került az út végére.

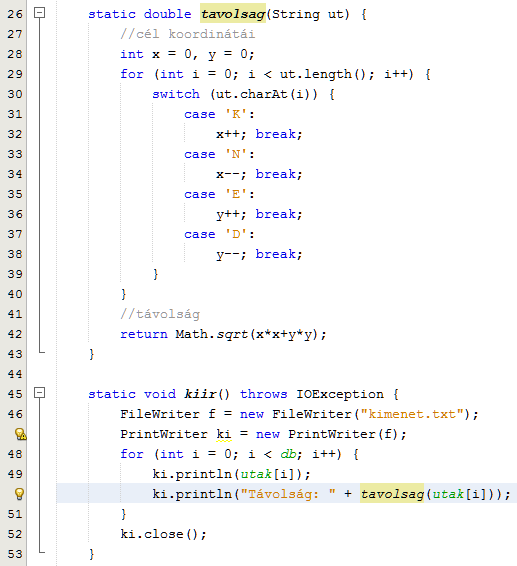
Ehhez először az utat végigjárva meghatározzuk az út végpontjának koordinátáit. Utána Pithagorasz-tétel segítségével kapjuk meg a távolságot.



The Pythagorean theorem

Először oldd meg papíron az első úttal!  
Az eredményeket írd ide: x = , y = , d =

Szúrd be a tavolsag() metódust a kiírás elé, majd egészítsd ki a kiir() metódust a távolság kiírásával (50.sor)! A program többi része nem változik.



Futtasd a programot, majd ellenőrizd a *kimenet.txt* fájlt!

## Energia

A robot a mozgásához szükséges energiát egy beépített akkuból nyeri. A robot 1 lépés megtételéhez 1 egység, az irányváltásokhoz és az induláshoz 2 egység energiát használ. Ennek alapján az EKK utasítássor végrehajtásához 7 egység energia szükséges.

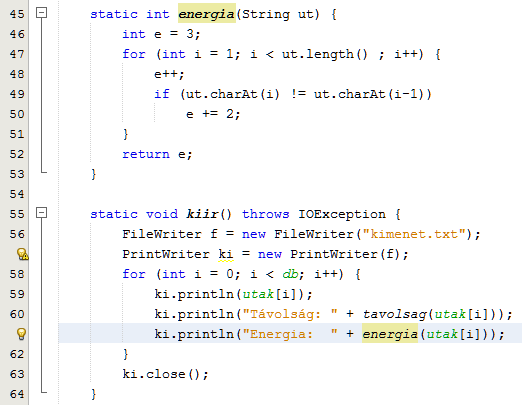
Mennyi energia kell a DND utasítássor végrehajtásához? Írd ide:

Alakítsuk át a programot úgy, hogy az eredeti utak és a távolságok után írja ki a szükséges energia mennyiségét is!

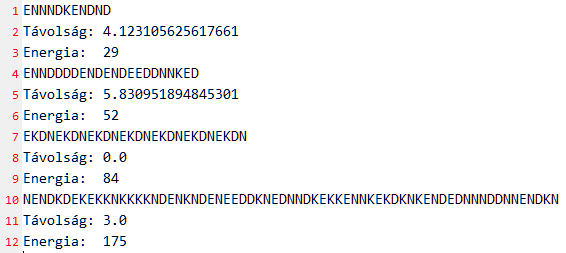
Az első lépéshez 3 egység kell (elindulás + 1 lépés). A további lépésekhez is kell 1 egység, és még 2 egység, ha irányváltás volt. Akkor történt irányváltás, ha a lépés nem egyezik meg az előtte lévővel. (Ezt csak a második, vagyis az 1-es indexű elemtől kezdődően tudjuk vizsgálni!)

Feltételeztük, hogy az út legalább egy lépést tartalmaz.

Szúrd be az energia() metódust a kiírás elé, majd egészítsd ki a kiir() metódust a szükséges energia kiírásával (61.sor)! A program többi része változatlan.



Futtasd a programot, majd ellenőrizd a *kimenet.txt* fájlt!



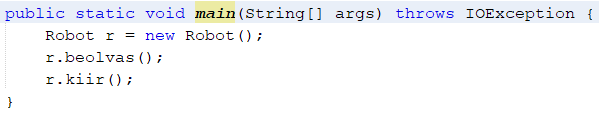
## Kiegészítés

Eddigi programjaink csak egy osztályból állnak, amely a main metódust is tartalmazza. Ilyen esetben az osztály minden változójának és metódusának statikusnak kell lennie.

Van azonban más megoldás is, amelyet gyakran alkalmaznak:

* A main metódusban létrehozunk egy példányt az osztályból. Ebben már nem fog lefutni a main metódus.
* A main-ből a példánynévvel együtt hívjuk meg a többi metódust. (pl. r.beolvas(); )
* Így az osztály változóinak és metódusainak nem kell statikusnak lenniük.

Próbáld ki ezt a Robot osztályban!

1. Alakítsd át a main metódust így:  
   
2. A main metódus kivételével mindenhonnan töröld a static szót!
3. Futtasd a programot, és ellenőrizd a kimenő fájlt!

## Önálló feladat

Kezeld a beolvas és a kiir metódusokban a keletkező kivételeket, majd távolítsd el a felesleges throws elemeket és close utasításokat!

# 17. Fájlok 5.

## Caaesar kódolás

Julius Caesarnak tulajdonítják a következő titkosírási módszert. Lényege, hogy minden karaktert az utána következő valahányadik (például harmadik) karakterrel helyettesítünk.

Ha az eltolás túlmutat az ábécé végén, az ábécé elejéről vesszük a karaktereket. Ehhez a kapott kódból le kell vonni az ABC hosszát (26).

Például a +3-as eltolás:

ABCDEFGHIJKLMNOPQRSTUVWXYZ  
ABCDEFGHIJKLMNOPQRSTUVWXYZABC

A kódolásban a fenti táblázat segít: megkeressük a karaktert a felső sorban, és helyettesítjük az alatta lévővel.

Példa: ABLAK 🡪 DEODN

Mi lesz a kódja a POGANY szónak? Írd ide:

Az üzenet olvasásakor ellenkező irányba kell eltolni a karaktereket ugyanannyival (vagyis -3-as eltolás). Ha az ABC elé kerülünk, a kódhoz hozzá kell adni az ABC hosszát.

ABCDEFGHIJKLMNOPQRSTUVWXYZ  
XYZABCDEFGHIJKLMNOPQRSTUVWXYZ

Mit jelent a kódolt VLPRQBL szó? Írd ide:

Ez a kódolás még könnyebb, mint az ATBAS módszer. Miért jobb mégis? Azért, mert nem csak 3-mal lehet eltolni a karaktereket, hanem tetszőleges 1 és 25 közötti számmal. Így a megfejtőnek több lehetőséget kell végigpróbálnia.

A mai leckében egy olyan programot készítünk, amely a *vers.txt* fájl tartalmát kódolja Caesar kódolással.

## Előkészületek

Kezdj egy új projektet *kodolas* néven!

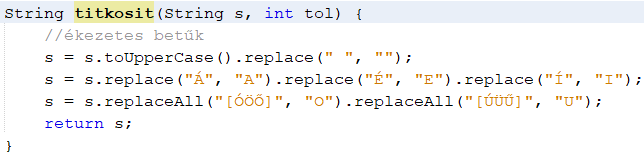
A Kodolas osztályban lesz a fájlkezelés, de magát a Caesar kódolást egy külön osztályban készítjük el. Ennek az az előnye, hogy a kódolást végző osztály másik programban is felhasználható.

Készíts egy új, Caesar nevú osztályt a kodolas csomagban! (jobb kattintás a csomag nevére, New, Java Class, Class name: Caesar, Finish)

## Titkosítás

Először a Caesar osztályt fogjuk elkészíteni. Egy titkosit nevű metódust hozunk létre benne, amely a fentiek szerint titkosítja a kapott szöveget. Paraméterei: s – a titkosítandó szöveg, tol – az eltolás mértéke.

A kódoláshoz először csupa nagybetűsre alakítjuk a szöveget, majd helyettesítjük az ékezetes betűket ékezet nélküli változatukkal. Korábban ezt egy elágazással oldottuk meg, de most egy másik módszert próbálunk ki. Nézzük a titkosit() metódus első változatát:



A korábban már tanult replace() metódust alkalmazzuk az ékezetes karakterek lecserélésére. Mivel több ilyen csere is kell, egymás után többször hívjuk meg a replace() metódust. Ezeket egymás után is fűzhetjük.

Van olyan eset, ahol többféle karaktert is ugyanazzal a karakterrel kell helyettesíteni. Ilyenkor a replaceAll() metódust alkalmazhatjuk. Ez úgy működik, hogy a []-ek közötti bármelyik karaktert lecseréli a megadottal.

Most ki kellene próbálni, hogy a metódus jól működik-e. Ezt megtehetnénk úgy, hogy a Kodolas osztály main metódusából meghívjuk valamilyen tesztadattal. Ezt az utasítást később törölnünk kellene.

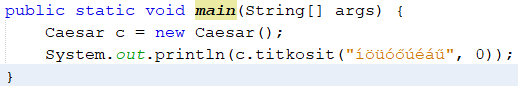
Ennél jobb megoldás az, hogy a Caesar osztályban is készítünk egy main metódust, és ebben helyezzük el a titkosit metódus teszteléséhez szükséges utasításokat.

A Caesar osztályt külön is futtathatjuk (Shift+F6 vagy jobb kattintás, Run file), és akkor a tesztet végezzük el a saját main metódusának végrehajtásával.

Ha viszont egy példányt hozunk létre a Caesar osztályból, akkor nem fogja végrehajtani a Caesar osztályban lévő main metódust.

A teszteléshez először létrehozunk egy példányt a Caesar osztályból, majd meghívjuk a titkosit metódusát egy teszt szöveggel, és kiíratjuk az eredményt.

Készíts egy main metódust a Caesar osztályba (psvm, majd Tab) az alábbi utasításokkal:

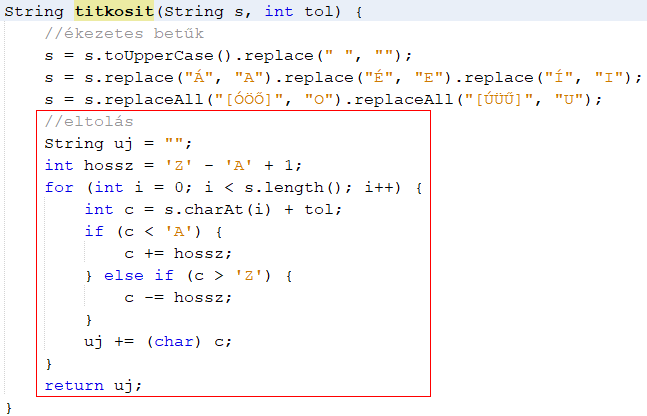


Figyeld meg, hogy a Caesar osztályban hoztunk létre egy új példányt saját magából! Ebben a példányban már nem hajtja végre a main metódust, és a titkosit metódusnak nem kell statikusnak lennie!

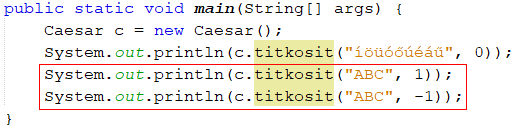
A titkosit metódus következő változatában már az eltolást is elvégezzük. Ennél kihasználjuk azt, hogy a Java szükség esetén automatikusan átalakítja a char típusú értéket int-té, és ez vissza is alakítható a (char) típuskonverzióval.

Egy uj nevű stringben állítjuk össze a kódolt szöveget. Karakterenként végigmegyünk az eredeti szövegen, és minden karakter kódjához hozzáadjuk az eltolást. Utána szükség esetén eltoljuk még az ABC hosszával is.

Egészítsd ki a titkosit metódust így:



Ezután egészítsd ki a main metódust is az eltolás tesztelésével:



Próbáld ki!

Ha jól működik, akkor a Caesar osztályt már használhatjuk a programban.

## Fájlkezelés

Vannak esetek, amikor szükség van arra, hogy a szöveges fájl sorait beolvassuk egy tömbbe, mert az egyes sorokkal többször is végeztünk műveleteket. Ezt akkor tudjuk elvégezni, ha tudjuk előre a sorok számát.

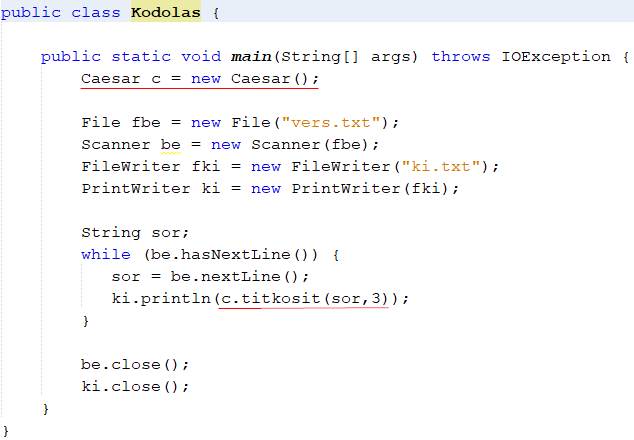
A gyakorlatban viszont sokszor nem tudjuk előre, hogy hány sor lesz. Ilyenkor három lehetőségünk van:

* Az egyik, hogy akkora tömböt hozunk létre, ami biztosan elég lesz, és egy változóban számláljuk az elemeket. Ennek a megoldásnak az a hátránya, hogy sok memóriát foglal.
* A második lehetőség, hogy a beolvasott adatokat egy listában tároljuk. Ennek nem kell előre megadni a méretét.
* A harmadik lehetőség, hogy amikor egy sort beolvasunk, azt rögtön feldolgozzuk, és utána nem tároljuk tovább. Így sokkal kevesebb memóriára van szükség, de ez csak akkor alkalmazható, ha a feldolgozást egyszerre el lehet végezni.

Mi most a harmadik megoldást fogjuk alkalmazni.

A programban először létrehozunk egy példányt a Caesar osztályból. Utána beolvassuk a sorokat a *vers.txt* fájlból, majd kiírjuk a *ki.txt* fájlba. Mindkét fájl UTF-8 kódolású.

Lássuk a programot:



A program elején megnyitjuk a bemeneti és a kimeneti fájlt is, a végén pedig bezárjuk őket. Ez különösen a kimeneti fájlnál fontos, mert enélkül lehet, hogy üres lesz!

A középső részben beolvassuk a sorokat, átalakítjuk őket, majd kiírjuk a kimeneti fájlba.

Másold a *vers.txt* fájlt a projekt mappájába, majd futtasd a programot! Ellenőrizd a *ki.txt* fájlt!

Változtasd az eltolást más értékre, és futtasd le így is!

## Kérdés

Hogyan lehetne visszaállítani az eredeti verset? Próbáld ki!

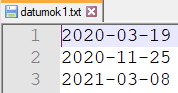
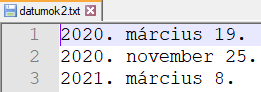
## Önálló feladat

Készítsd el a hibakezelést try – catch szerkezettel!

# 18. Fájlok 6.

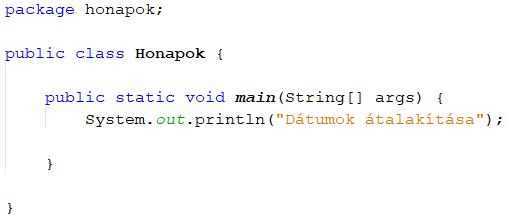
Ebben a leckében egy olyan programot készítünk, amely dátumokat alakít át.

A dátumokat egy szöveges fájlban kapjuk (datumok1.txt), és átalakítás után egy másik szöveges fájlba kell írnunk (datumok2.txt). Figyeld meg a dátumformátumokat!

## Új program

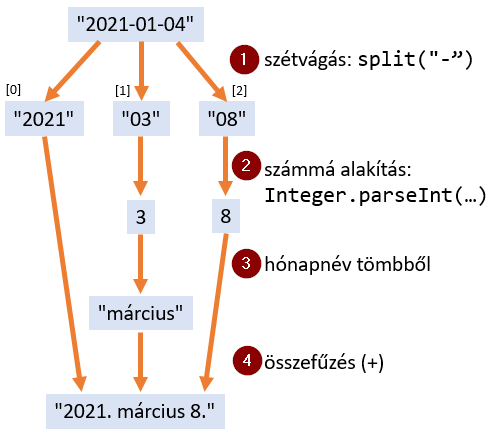
Készíts egy programot honapok néven, és írasd ki a program címét!



## Átalakítás

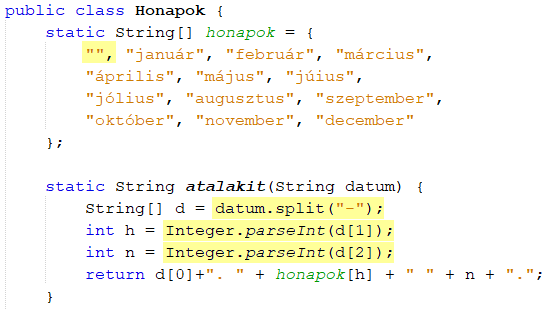
Először egy metódust fogunk készíteni, amely egy dátumot alakít át.

Gondoljuk először végig, hogyan lehet ezt elvégezni:



1. A dátumot elször szétvágjuk három részre a kötőjeleknél (-). Ehhez a split metódust használjuk. Paraméterként azt a karaktert adjuk meg, amelynél a vágásokat el kell végezni. A szétvágás után a dátum részei egy tömbbe kerülnek.
2. A tömb 1-es és 2-es indexű elemét átalakítjuk számmá. Ezt legegyszerűbben az Integer.parseInt metódussal végezhetjük el. Ez egy stringet kap paraméterként, és egy ebből átalakított egész számot ad vissza.
3. Az egyik kapott szám a dátum hónapjának sorszáma. A hónap nevének meghatározásához készítünk egy tömböt a hónapok neveivel, és ebből vesszük a megeflelő elemet. (Mint az előző feladatban, amelyben a számok neveit olvastuk ki egy tömbből.)  
   A napot azért alakítottuk számmá, hogy eltávolítsuk a 0-t az elejéról.
4. Végül összefűzzük a dátum részeit a megfelelő karakterekkel elválasztva. Az összefűzésnél a számok automatikusan stringgé alakulnak.

Írd az alábbiakat a main metódus elé:

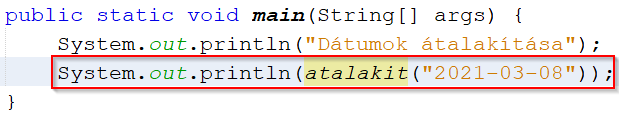


A honapok tömbnél alkalmaztunk egy kis trükköt (sárgával kiemelve): A nulladik elembe felvettünk egy üres stringet.

Ez azért jó, mert így a hónapok éven belüli sorszáma és a tömbön belüli indexe megegyezik. Például a 3-as indexű hónap március lesz. A nulladik elemet nem fogjuk használni.

Figyeld meg, hogyan vágjuk szét a dátumot részekre, és hogyan alakítjuk át a hónap számát stringből egész számmá! (Ezeket is kiemeltük sárgával.)

A metódus teszteléséhez írd a main metódusba a következőket:



Futtasd a programot! Ellenőrizd az eredményt!

## Fájlok megnyitása

A programnak a kapott fájlban lévő dátumokat kell beolvasnia és átalakítás után kiírnia egy másik fájlba. Ezért a main metódust át kell írnunk.

Első lépésben megnyitjuk a bemenő és a kimenő fájlt.

A bemenő fájlt egy Scanner segítségével olvassuk, a kimenetre pedig egy PrintWriter-rel írunk.

A fájlok megnyitását egy try-catch szerkezettel végezzük.

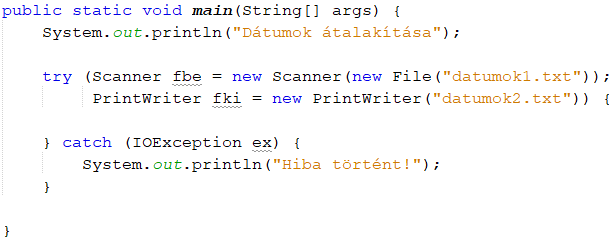
A try után zárójelek között, pontosvesszővel elválasztva adjuk meg azokat az utasításokat, amelyek megnyitják a fájlokat.

A kapcsos zárójelután a try ágba írjuk majd azokat az utasításokat, amelyek elvégzik a beolvasást és a kiírást.

A végén a program automatikusan bezárja a fájlokat.

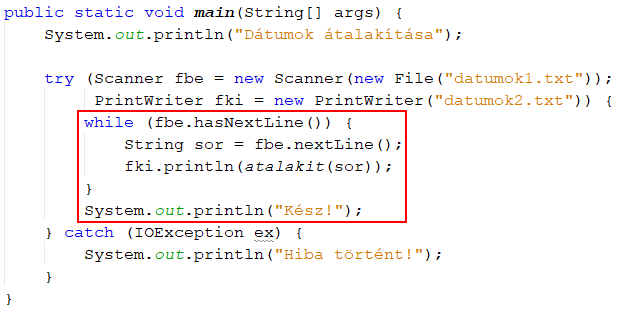
Hiba esetén (ha például nem találja a bemenő fájlt) a catch ágban lévő utasítások futnak le.

Írd át a main metódust így:



## Beolvasás, kiírás

A bemenő fájlból addig olvasunk, amíg van következő sor. A beolvasott sort átalakítjuk az atalakit metódussal, majd kiírjuk a kimenő fájlba. Egészítsd ki a main metódust az alábbiak szerint:



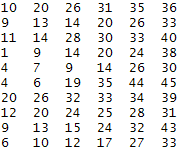
Másold a kapott fájlt a projekt mappájába, majd futtasd a programot! Ellenőrizd a kimenő fájlt!

# 19. Fájlok 7.

## Lottó

Ebben a leckében a hatos lottóval kapcsolatos statisztikákat fogunk készíteni.

Az UTF-8 kódolású *hatos.txt* fájl a hatos lottó kihúzott számait tartalmazza 2005. januárjától 2012. júliusáig. Soronként vannak egy-egy hét számai (növekvő sorrendben, tabulátorokkal elválasztva):



A sorok száma 395. A számokat egy olyan kétdimenziós tömbben szeretnénk tárolni (szamok), amely 400 soros (hátha lesznek még adatok), és minden sorban 6 egész szám van.

A beolvasott sorok számát is tárolni kell, erre létrehozunk egy db nevű, egész típusú változót.

Mivel ezeket a változókat minden metódusból el kell tudni érni, az osztály elején statikus változóként adjuk meg őket.

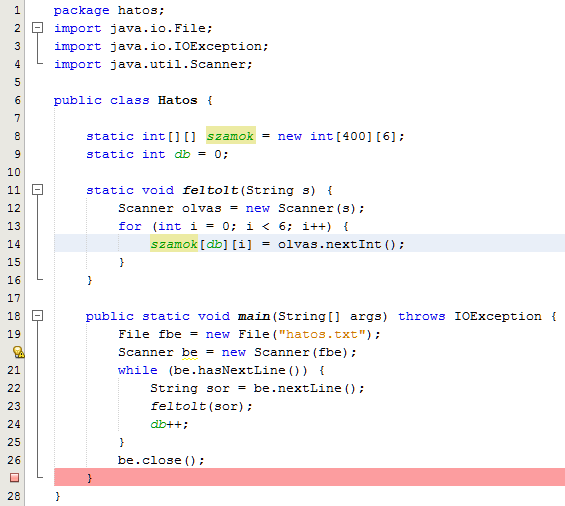
## Beolvasás

A program első változata beolvassa az adatokat a fájlból, és eltárolja őket a tömbben.

A szöveges fájl sorait a szokott módon olvassuk be, de a sorok feldolgozását a feltolt() metódussal végezzük.

Ebben létrehozunk egy másik Scanner objektumot, amely a kapott stringből olvas úgy, mintha a billentyűzetről írnák be a számokat. A beolvasott számokat elhelyezi a tömb megfelelő elemében.

Lássuk a programot! A törésponttal egyelőre ne foglalkozz, mindjárt lesz róla szó.



Készítsd el, és futtasd le a programot! Ne felejtsd el előtte a program mappájába másolni a hatos.txt fájlt!

## Ellenőrzés

A program futtatásakor a kimeneten nem jelenik meg semmi. Jó lenne megnézni, mi került a szamok tömbbe, de elég nehézkes lenne kiíratni az elemeket.

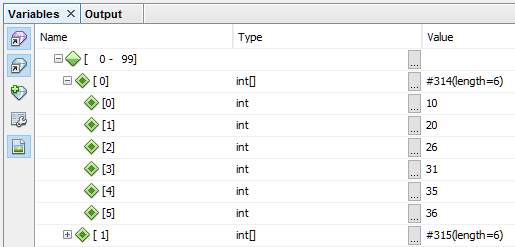
Ehelyett a hibakeresőt (debugger) fogjuk alkalmazni. Ennek indításához a Debug Project gombot vagy a Ctrl+F5 billentyűket kell megnyomni. Ha kipróbálod, akkor azt fogod tapasztalni, hogy a program ugyanúgy végigfut, mint eddig.

Meg kellene állítani a programot, és akkor megnézni a változók értékeit. A program megállítására szolgálnak a töréspontok.

Töréspontot a sor előtti sorszámra kattintva lehet elhelyezni vagy törölni. Helyezz el most egy töréspontot a 27. sorban! A rózsaszín sáv és a sorszám helyén a négyzet jelzi a bekapcsolt állapotot. Azért ebbe a sorba tettük, mert ez már az adatok beolvasása után van.

Indítsd el most a debug módot a Ctrl+F5 megnyomásával! A program lefut a 26. sorig, majd megáll. A NetBeans ablak alsó részén, a kimenetnél megjelenik a Variables fül. Válts át rá!

Kattints a static, majd a szamok előtti pluszra! A szamok tömbben nagyon sok érték van, ezért százanként csoportosítva jelennek meg. Jelenítsd meg a szamok[0]-ban lévő elemeket!



Vesd össze a számokat a *hatos.txt* első sorával! Ha mindent jól csináltál, akkor ezek megegyeznek. Ugyanígy végig lehetne nézni a többi sort is, de nagyon sokáig tartana. A biztonság kedvéért azért ellenőrizd az utolsó sort és az elemek számát (db)!

Az ellenőrzés után futtasd tovább a programot a zöld körben lévő nyíl vagy az F5 megnyomásával! Mivel nincs több utasítás, meg fog állni.

Végül töröld a töréspontot!

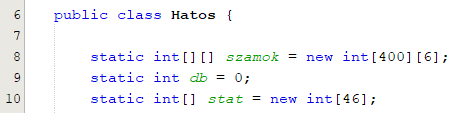
## Statisztika

Egy olyan statisztikát szeretnénk készíteni, amelyből kiderül, hogy melyik számot hányszor húzták ki az évek során. Ez egy kicsit hasonlít arra a feladatra, amikor nekünk kellett generálni a számokat.

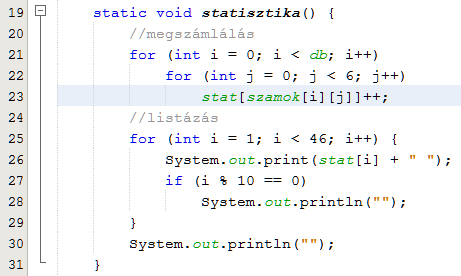
Ott egy olyan tömbbel modelleztük a szelvényt, amelynek minden eleme boolean típusú volt, és azt adta meg, hogy be van-e jelölve az adott szám.

Most egy olyan tömböt fogunk készíteni, amelynek minden eleme egy egész szám, és azt jelzi, hogy hányszor húzták már ki az adott számot.

A hatos lottón 45 számból húznak. A tömb 46 elemű lesz, a 0. elemet nem fogjuk használni.



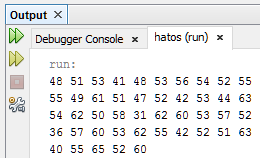
A statisztikát a statisztika() metódussal fogjuk elkészíteni. A metódus első felében megszámoljuk, melyik számot hányszor húzták ki, a második felében pedig kilistázzuk az eredményeket:



A szamok tömb elemein két for ciklussal tudunk végigmenni. A stat tömbben mindig azt az elemet növeljük eggyel, amelynek a szamok tömb adott eleme az indexe. Ha például a szamok i-dik sorának j-dik oszlopának értéke 23, akkor a stat[23]-hoz adunk hozzá egyet.

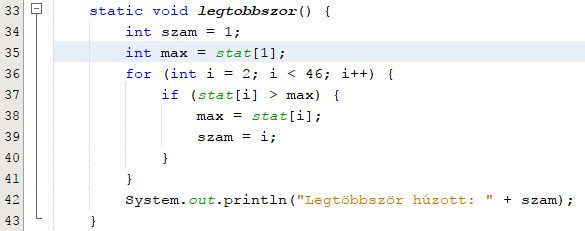
A listázásnál minden 10. elem után és a legvégén kiíratunk egy soremelést is.

Írd be a kódot és egészítsd ki a main() metódust a statisztika() metódus hívásával, majd próbáld ki a programot! Hogyan lehet értelmezni a kimenetet?

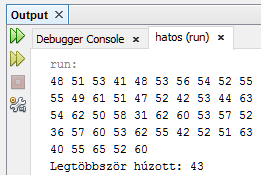


## Legtöbbször húzott szám

Ki szeretnénk íratni, hogy melyik számot húzták ki a legtöbbször. Ehhez meg kell keresnünk a stat tömb legnagyobb értékét, és ennek az indexét kell kiíratnunk:



Írd be a metódust, egészítsd ki a main() metódust a legtobbszor() metódus hívásával, majd próbáld ki a programot!

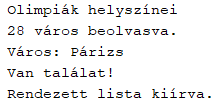


## Feladat

Egészítsd ki a programot úgy, hogy azt is írja ki, melyik számot húzták ki a legkevesebbszer! Melyiket?

# 20. Fájlok 8.

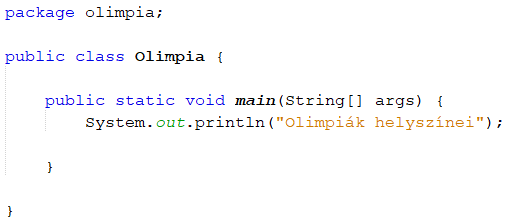
Ebben a leckében beolvassuk egy fájlból (olimpia1.txt) az eddigi olimpiák helyszíneit, megnézzük, hogy egy megadott város szerepel-e közöttük, majd ABC sorrendben kiírjuk egy másik fájlba (olimpia2.txt).



   
…

## Új program

Készíts egy programot olimpia néven, és írasd ki a program címét!



## Beolvasás

Most nem csak egyszer kell végigmenni az adatokon, mint az előző feladatban. Ezért az adatokat beolvassuk a memóriába, és egy listában (ArrayList) tároljuk. Ez abban különbözik a tömböktől, hogy mérete automatikusan változik. Így a szöveges fájl hossza tetszőleges lehet.

A listák kezelését az alábbi metódusokkal végezhetjük el:

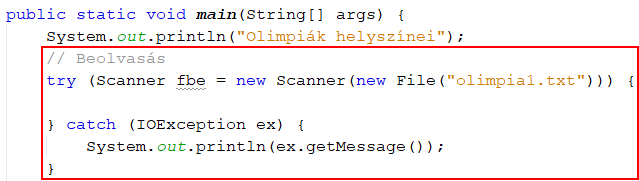
* add(s): új elem hozzáadása a lista végéhez,
* get(i): i-dik elem a listából,
* size(): a lista hossza (az elemek száma).

Először létrehozzuk a listát, amelynek elemei stringek lesznek.

Írd ezt az osztály elejére:



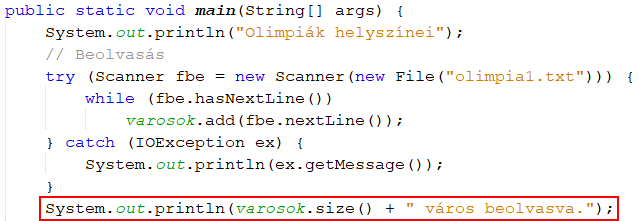
A bemenő fájl megnyitását try…catch szerkezettel végezzük:



A bemenő fájl megnyitása után egy ciklusban beolvassuk a sorokat, és betesszük őket a lista következő elemébe:



A beolvasás után kiíratjuk a lista méretét, amely megegyezik a beolvasott városok számával:



Másold az olimpia1.txt fájlt a projekt mappájába, majd futtasd a programot! Ellenőrizd, hogy a beolvasott sorok száma megegyezik-e a szöveges fájlban lévő sorok számával!

## Keresés

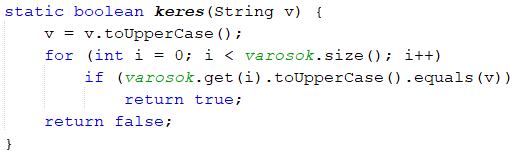
Ezután a program kérjen egy városnevet, és írja ki, van-e ilyen város a listában!

Ehhez először készítünk egy külön metódust keres néven. Ez a metódus kap egy városnevet paraméterként, amelyet nagybetűsre alakít.

Ezután egy ciklus segítségével sorban veszi a listában lévő városokat, szintén nagybetűsre alakítva. (Így mindegy, hogy a felhasználó kis- vagy nagybetűvel adta meg a városnevet.)

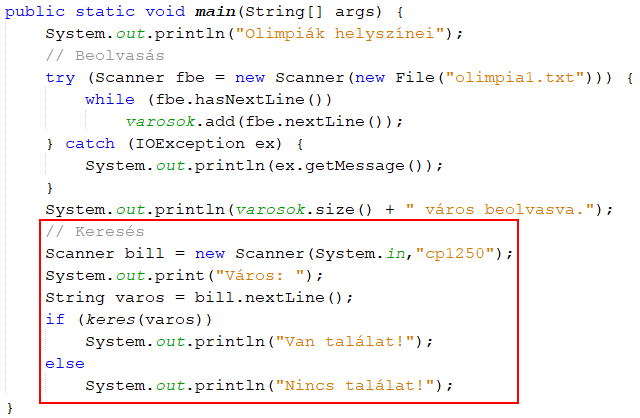
Ha egy város neve megegyezik a keresettel, akkor kilép a ciklusból és a metódusból, és igaz értéket ad vissza. (Nem keres tovább.)

Ha a lista végére ért, és nem talált egyezést, akkor hamis értéket ad vissza. Írd be a metódust:



Mivel a varosok változót a metódusokon kívül deklaráltuk, az a keres és a main metódusban is használható.

Írd be a fenti metódust, majd egészítsd ki a main metódust a következőkkel:



Próbáld ki olyan várossal, amely szerepel a fájlban, és olyannal is, amely nem! Próbáld ki kis- és nagybetűkkel is!

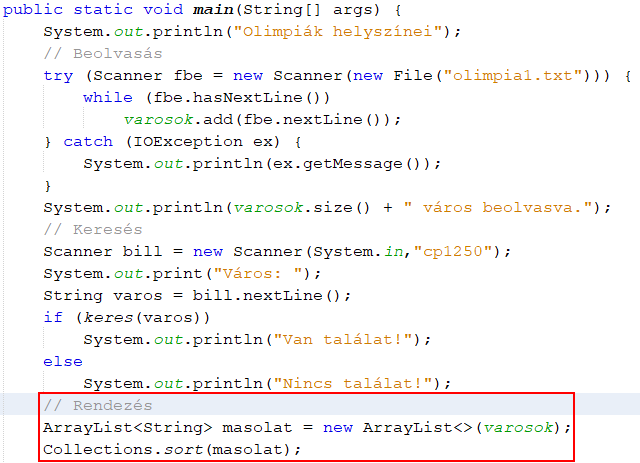
## Rendezés

A listák rendezéséhez a Collections osztály sort metódusát használhatjuk.

Ez a metódus osztálymetódus, vagyis híváskor az osztály nevét (Collections) kell elé írni. (És a Collections osztályt be kell importálni.)

Az eredeti varosok listát meg szeretnénk őrizni, ezért készítünk róla egy másolatot, amelyet rendezünk. A masolat egy új lista lesz, amit a varosok listával inicializálunk.

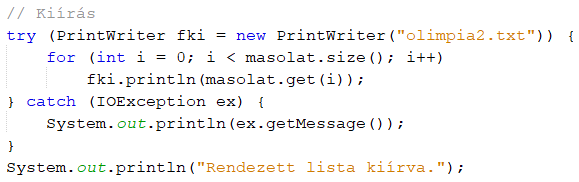
Egészítsd ki a main metódust az alábbiak szerint:



Mivel a masolat listát a main metódusban deklaráltuk, csak azon belül használható!

## Kiírás

Végül kiírjuk a masolat lista elemeit az olimpia2.txt fájlba. Írd az alábbiakat a main metódus végére:



Először megnyitjuk a kimenő fájlt egy PrintWriter-rel.

A fájlok megnyitását egy try-catch szerkezettel végezzük. A try után zárójelek között nyitjuk meg a PrintWriter-t. A kapcsos zárójel után a try ágban írjuk ki az adatokat a kimenő fájlba. A végén a program automatikusan bezárja a fájlokat.

Hiba esetén (ha például nem tud írni a kimenő fájlba) a catch ágban kiíratjuk a hibaüzenetet.

Futtasd a programot! Ellenőrizd a kimenő fájlt!

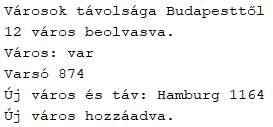
# 21. Osztályok 1.

Ebben a leckében beolvassuk egy fájlból (tavolsag.txt) néhány európai város Budapesttől mért távolságát.

Ezután bekérünk egy városnév részletet, majd kilistázzuk azoknak a városoknak az adatait, amelyekben a megadott részlet szerepel.

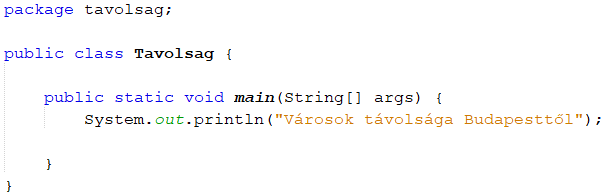
Végül bekérjük egy új város adatait, és hozzáfűzzük a tavolsag.txt fájl végéhez.





## Új program

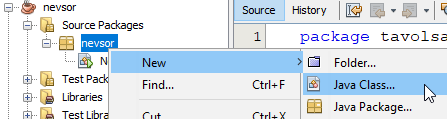
Készíts egy programot tavolsag néven, és írasd ki a program címét!



## Új osztály készítése

Minden városnak van neve (szöveg) és távolsága (egész szám). Ezeket együtt szeretnénk tárolni, és műveleteket is végzünk velük. Ezt egy osztály segítségével tudjuk megtenni.

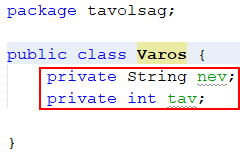
Készíts egy új osztályt Varos néven:



Ebből az osztályból fogunk példányokat (objektumokat) létrehozni minden városhoz. Az objektumok típusa Varos lesz.

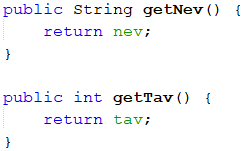
Az osztályban azt kell megadni, hogy egy városnak milyen adatai vannak (változók/mezők/adattagok), és hogy milyen műveleteket lehet végezni velük (metódusok).

Kezdjük az adatokkal:

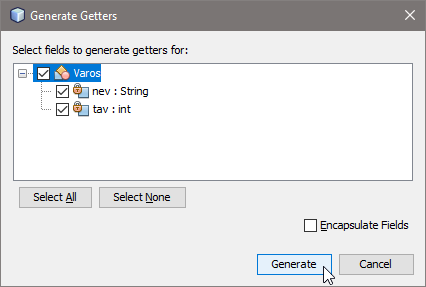


Az adatok előtti private szó azt jelzi, hogy ezek a változók más osztályokból nem érhetők el. Ezzel védjük az adatokat a kívülről történő módosításoktól.

A Tavolsag osztályban azonban szükségünk lesz a névre és a távolságra, ezért készítünk hozzájuk lekérdező metódusokat. Ezek neve általában get-tel kezdődik (de ez nem kötelező). A public szó jelzi, hogy ezeket a metódusokat más osztályokból is meghívhatjuk.



A lekérdező metódusokat a NetBeans automatikusan megírja nekünk, ha megnyomjuk az Alt+Insert billentyűket, majd a Getters parancsot választjuk:



Minden osztályban lehetnek publikus setter metódusok is, amelyekkel beállítjuk egy példányváltozó értékét. Ezek neve általában set-tel kezdődik.

Például lehetne most egy setNev metódus, amellyel a nev értékét állítanánk be, de erre most nincs szükségünk.

Az adatokat setter metódusok helyett a konstruktor segítségével az objektumok létrehozásakor fogjuk beállítani.

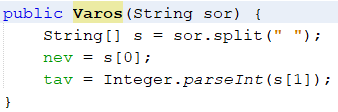
Amikor beolvasunk egy sort a szöveges fájlból, akkor abból a sorból kell egy új várost létrehozni.

Ehhez írunk egy konstruktort az osztályba, amely a példány létrehozásakor automatikusan lefut.

A konstruktor neve mindig megegyezik az osztály nevével, és sosem adjuk meg a visszaadott érték típusát. Paraméterként most megkap egy beolvasott sort.

A sort először szétvágjuk a szóköznél. Az első fele lesz a város neve. A második felét egésszé alakítjuk, és a tav változóba tesszük.

Írd be a konstruktort:



Mentsd az osztályt! Kipróbálni később fogjuk.

## Beolvasás

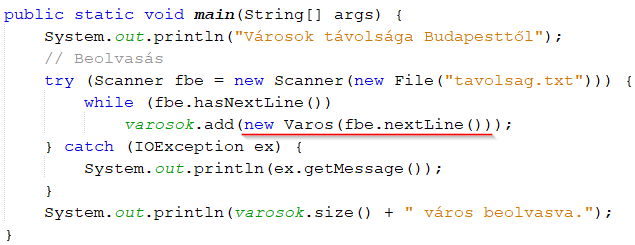
Az adatokat beolvassuk a fájlból a memóriába, és egy listában (ArrayList) tároljuk, amelynek mérete automatikusan változik. Így a szöveges fájl hossza tetszőleges lehet.

A lista elemei városok lesznek (vagyis a Varos osztály példányai).

Írd ezt a Tavolsag osztály elejére:



A bemenő fájl megnyitását try…catch szerkezettel végezzük:



Figyeld meg, hogy most nem a beolvasott sort adjuk hozzá a listához, hanem a belőle készített új, Varos típusú objektumot! (piros aláhúzással jelölve) A Varos osztály konstruktorát a beolvasott sorral hívjuk meg.

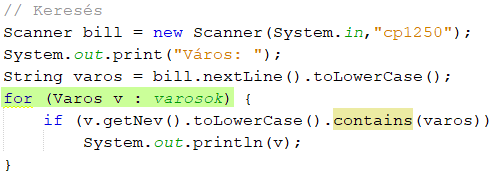
A beolvasás után kiíratjuk a lista méretét, amely megegyezik a beolvasott városok számával.

Másold a tavolsag.txt fájlt a projekt mappájába, majd futtasd a programot! Ellenőrizd, hogy a beolvasott sorok száma megegyezik-e a szöveges fájlban lévő sorok számával!

## Keresés

Ezután a program kérjen egy városnév részletet, és írja ki azoknak a városoknak a nevét és távolságát, amelyekben a megadott részlet szerepel!

Folytasd így a main metódust:



A beolvasott névrészletet kisbetűsre alakítjuk.

A listán most egy for...each ciklussal megyünk végig (zölddel kiemelve). Ez a v változóba sorban beleteszi a lista összes elemét, és minddel végrehajtja a ciklust.

A ciklusban vesszük a város nevét kisbetűsre alakítva, és megnézzük, hogy tartalmazza-e (contains) a beolvasott városnév részletet. Ha a megadott részlet több város nevében is szerepek, akkor mindet kiíratjuk.

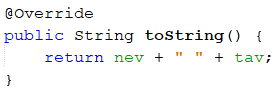
A contains metódus (sárgával kiemelve) akkor ad vissza igaz értéket, ha a bal oldalán lévő string tartalmazza a zárójelben megadott részletet.

Próbáld ki! Figyeld meg a kiírást! Ez így még nem jó.

Minden objektumnak van egy toString nevű metódusa. Ha az objektumot kiíratjuk a képernyőre vagy egy fájlba, akkor a program a toString metódust hívja meg.

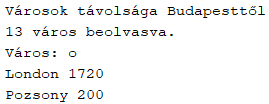
Az objektum toString metódusát átírhatjuk. Válts a Varos osztályra, az osztály kódjában nyomd meg az Alt+Insert billentyűket, és válaszd a toString… parancsot!

Ezután a kapott metódust írd át így:



A @Override annotáció azt jelzi, hogy felülírunk egy már létező metódust.

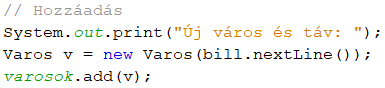
Futtasd a programot, és figyeld meg a kiírást!



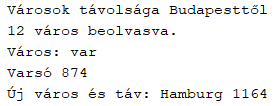
## Hozzáadás

Ezután beolvassuk egy új város adatait, és hozzáfűzzük a varosok listához és a tavolsag.txt fájl végéhez is.

Először olvasd be az új város adatait, hozz létre egy új objektumot belőlük, és add hozzá a lista végéhez:

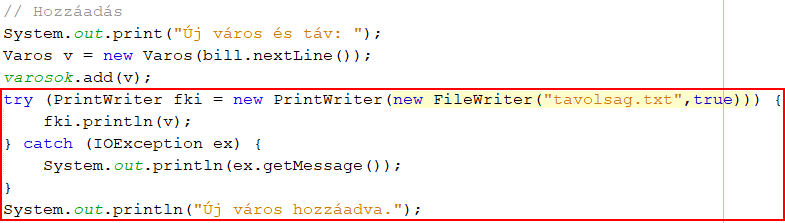


Feltételezzük, hogy jól írják be az adatokat. Próbáld ki!



Ezután a tavolsag.txt fájl végéhez szeretnénk hozzáfűzni az új város adatait úgy, hogy a többi adat is megmaradjon.

Fejezd be így a main metódust:

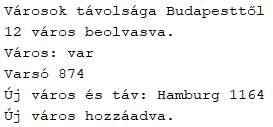


Figyeld meg, hogy a PrintWriter-t nem a szokott módon hoztuk létre, hanem először egy FileWriter-t készítettünk, és abból egy PrintWriter-t (sárgával kiemelve).

A FileWriter konstruktorának true paramétere jelzi, hogy a fájl korábbi tartalmát meg kell őrizni, és az új adatokat a fájl végéhez kell hozzáfűzni.

Ha a PrintWritert a szokott módon nyitnánk meg, akkor a korábbi adatok elvesznének.

Futtasd a programot! Ellenőrizd a kimenő fájlt!



# 22. Osztályok 2.

## Feliratok

A videókhoz a legtöbb lejátszóprogram meg tud jeleníteni feliratokat. A feliratokat egy külön srt kiterjesztésű feliratfájlban szokták megadni, amely tartalmazza a feliratok sorszámát, az időzítéseket és a feliratokat.

Ebben a feladatban egy videóhoz SRT formátumú, angol nyelvű feliratfájlt kell készítened. A feliratok és azok időzítései rendelkezésre állnak a *feliratok.txt* állományban, de a formátumuk nem megfelelő.

A *feliratok.txt* állományban 2-2 soronként ismétlődve egy felirat időzítése és az ehhez tartozó felirat található:

00:01 - 00:03  
So phase two - tank creation.

A példában a felirat a videó lejátszásakor az első másodpercnél jelenik meg ( 00:01 ), a harmadik másodpercben tűnik el ( 00:03 ), és a megjelenített felirat szövege a „So phase two - tank creation”.

(Forrás: <https://unity3d.com/learn/tutorials/projects/tanks-tutorial>)

## Osztály létrehozása

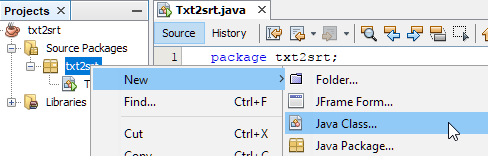
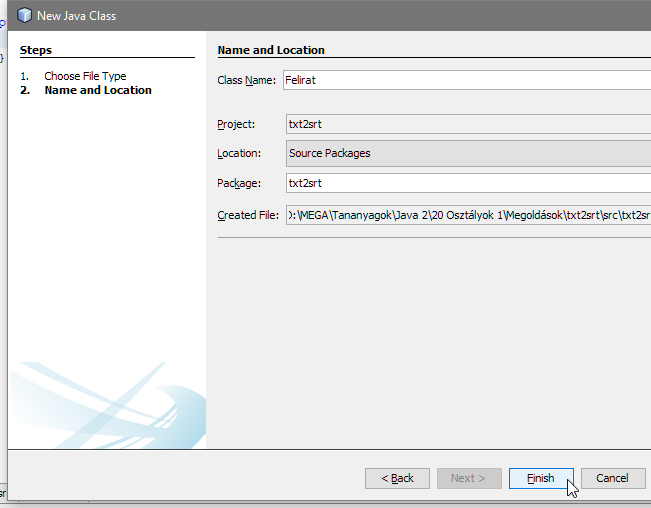
Először meg kell terveznünk, hogy milyen adatszerkezetben tároljuk az adatokat.

Minden feliratnál két idő adatra és egy szövegre van szükség. Az időket átszámoljuk másodpercekre, és így tároljuk.

Eddigi tudásunk alapján a tárolást megoldhatnánk három tömbbel, de van jobb megoldás is.

Minden időzített felirat adatait egy-egy objektumban fogjuk tárolni, és ezekből később létrehozunk egy listát. Az objektum típusa Felirat lesz. A Felirat osztály három mezőt fog tartalmazni: a felirat megjelenésének idejét (ido1), az eltűnés idejét (ido2) és a feliratot (szoveg).

A Java-ban minden osztályt külön fájlba szoktunk írni. Most is így fogjuk csinálni:

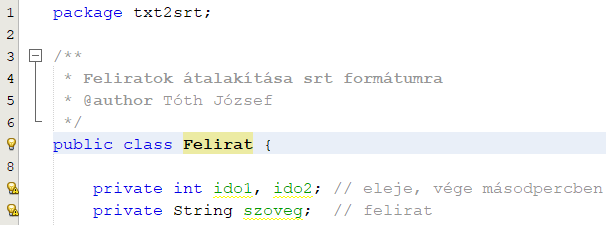
1. Először kezdj egy új projektet *txt2srt* néven a szokott módon!
2. A Projects ablakban kattints a jobb gombbal a txt2srt csomagra, majd a menüből válaszd a New, Java Class... parancsot!  
   
3. Az osztály neve legyen Felirat (nagy F-fel!), majd kattints a Finish gombra!  
   

Így most már két osztályból áll a programunk:

* A Felirat osztályban definiáljuk az egyes feliratok kezeléséhez szükséges adattagokat (mezőket) és metódusokat.
* A Txt2srt osztályban van a main() metódus, a program indításakor az fog elindulni.

## A Felirat osztály

A Felirat osztályban először megadjuk az adattagokat. Írd be a következőket: (A szerző nevéhez írd a saját nevedet!)



Az osztályoknak a tömbökkel szemben az az egyik előnyük, hogy különböző típusú adatokat is együtt lehet velük kezelni.

Mivel a Felirat osztály példányait használjuk majd, itt nem kell a static szó sem az adattagok, sem a metódusok elé.

Most már két osztályunk van, ezért el kell gondolkodnunk az adattagok és metódusok láthatóságán is. Ha nem jelezzük külön a public vagy a private szóval, akkor ezek egy csomagon belül minden osztályból elérhetők, módosíthatók.

Az osztályokat általában úgy írjuk meg, hogy védjék a saját adataikat: más osztályból csak ennek az osztálynak a metódusaival lehessen lekérdezni vagy módosítani őket (encapsulation). Ezért írtuk a private kulcsszót minden adattag elé.

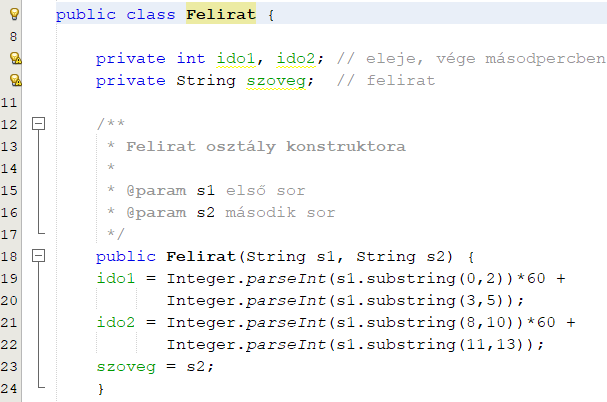
Ha a public kulcsszót írtuk volna egy adattag elé, akkor bármelyik csomag bármelyik osztályából elérhető lenne. Ez nem ajánlott.

Az osztályok metódusai között van egy különleges: a konstruktor. Amikor létrehozunk az osztály alapján egy új objektumot, akkor a Java automatikusan lefuttatja a konstruktort.

A konstruktort általában a kezdőértékek beállítására használjuk. Most például egy olyan konstruktort fogunk készíteni, amely két stringet kap, és ebből beállítja a három adattag értékét.

A konstruktor az időértékeket kiolvassa a kapott stringből, és átváltja másodpercekre.

Egészítsd ki a Felirat osztályt a konstruktorral:



Gondold végig a konstruktor működését!

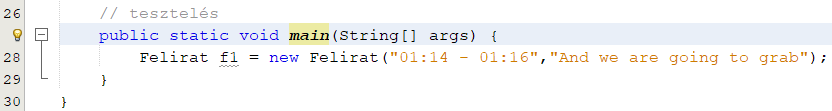
A konstruktor:

* kötelezően public (ami azt jelenti, hogy más osztályokból és csomagokból is látható),
* soha nem ad vissza értéket, ezért ennek típusát nem is adhatjuk meg (void sem lehet),
* neve mindig megegyezik az osztály nevével.

## Tesztelés

Mielőtt használnánk egy osztályt egy programban, célszerű tesztelni, hogy jól működik-e.

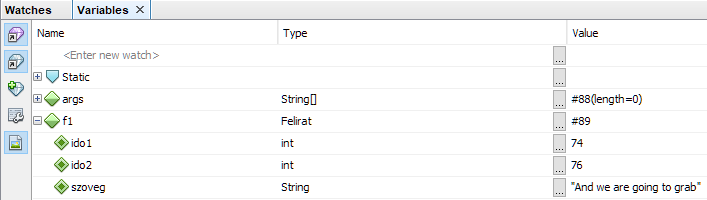
Ezt most úgy oldjuk meg, hogy készítünk egy main metódust a Feirat osztályba, és utána a Felirat.java fájlt futtatjuk. Ekkor ennek main metódusa fog elindulni, amelyben létrehozunk egy új példányt az osztályból.

1. A Felirat osztályban készíts egy main metódust a következőképpen:  
   

Az időzítést és a szöveget átmásolhatod a *feliratok.txt* fájlból.

Helyezz el egy töréspontot a 29. sorban, majd indítsd el a debuggert (Ctrl+Shift+F5, vagy jobb kattintás és Debug File)!

Ellenőrizd az f1 objektum adattagjait:



Megjegyzés: A Felirat osztályt úgy lehet futtatni, hogy megnyomod a Shift+F6 billentyűket, vagy a jobb kattintás után a Run File parancsot választod.

## További metódusok

Ezután el kell gondolkodni rajta, hogy milyen további metódusokra lesz szükség a Felirat osztályban. Mit gondolsz?

A következő leckében folytatjuk.

# 23. Osztályok 3.

## Feliratok

Ebben a leckében folytatjuk a legutóbb elkezdett felirat átalakító programot.

Emlékeztető: Ebben a feladatban egy videóhoz SRT formátumú, angol nyelvű feliratfájlt kell készítened. A feliratok és azok időzítései rendelkezésre állnak a *feliratok.txt* állományban, de a formátumuk nem megfelelő.

A *feliratok.txt* állományban 2-2 soronként ismétlődve egy felirat időzítése és az ehhez tartozó felirat található:

00:01 - 00:03  
So phase two - tank creation.

A példában a felirat a videó lejátszásakor az első másodpercnél jelenik meg ( 00:01 ), a harmadik másodpercben tűnik el ( 00:03 ), és a megjelenített felirat szövege a „So phase two - tank creation”.

(Forrás: <https://unity3d.com/learn/tutorials/projects/tanks-tutorial>)

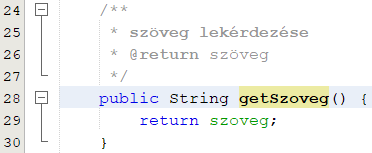
Nyisd meg a múltkor elkészített projektet, és nézd át az eddigi kódokat!

## Szöveg lekérdezése

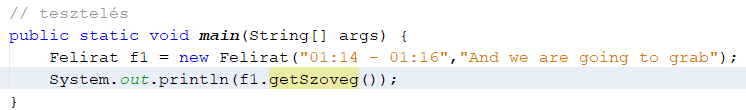
A szöveg adattagot privátra állítottuk, de a szöveg értékére szükségünk lesz. Ilyen esetekben készítünk egy publikus metódust, amely megadja a tárolt szöveget.

Az adattagok értékét lekérdező metódus neve általában get-tel kezdődik.

Egészítsd ki a Felirat osztályt ezzel a metódussal:



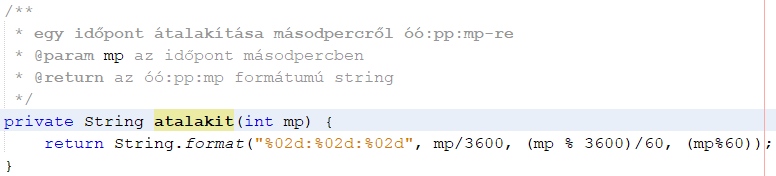
Ezt tesztelni is kell, ezért egészítsd ki a main metódust is! (A korábban elhelyezett töréspontot távolítsd el!)



Futtasd le a tesztet (Shift+F6)!

## Időzítés átalakítása

Szükségünk lesz egy metódusra, amely a másodpercben tárolt időt átalakítja óó:pp:mp formáttumra, mert erre lesz szükségünk. A metódust csak a Felirat osztályból hívjuk meg, ezért privát lesz:



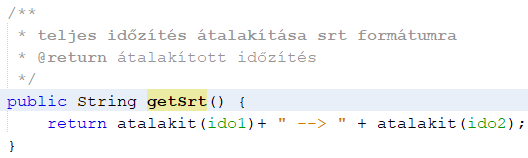
A String.Format() metódus a printf-hez hasonlóan működik, de az eredményt nem a képernyőre írja, hanem egy új stringben helyezi el.

A %02d helyére egy kétjegyű szám kerül. Ha csak egyjegyű lenne, akkor egy 0-t ír elé.

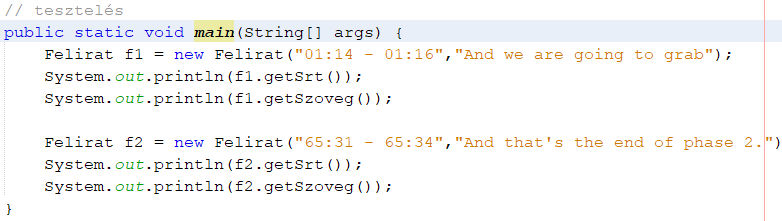
Gondold végig a metódus működését, majd írd be a Felirat osztályba!

Ezt a metódust majd a következő metódussal együtt teszteljük, amely meg fogja hívni ezt.

A getSrt metódus SRT formátumban fogja előállítani az időzítést az ido1 és az ido2 adatokból, az atalakit metódus segítségével:

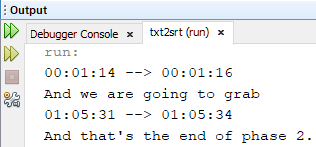


A teszteléshez egészítsd ki a main metódust!



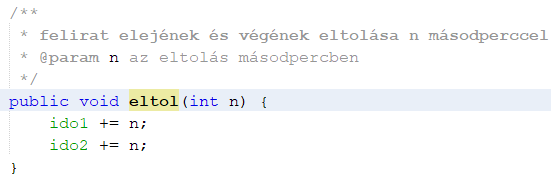
Most két felirattal is tesztelünk. A második 60 percnél nagyobb időértékeket tartalmaz.

Futtasd le a tesztet (Shift+F6)! Ezt kell kapnod:

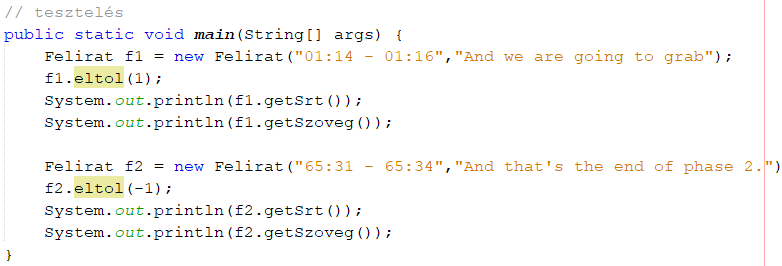


## Eltolás

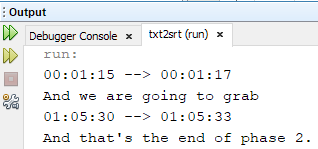
Előfordulhat, hogy a felirat elcsúszik a képhez viszonyítva. ezért hasznos egy olyan metódus, amellyel eltolhatjuk az időzítést:



Írd be, majd egészítsd ki a tesztet így:



Próbáld ki, és ellenőrizd az eredményeket!



## Dokumentáció

Ezzel elkészültünk a Felirat osztállyal, az más osztályokból már felhasználható. Ennek megkönnyítése érdekében elkészítjük az osztály dokumentációját.

A dokumentáció elkészítésének első és legfontosabb lépése, hogy a program kódjában elhelyezzük a dokumentáció szövegét.

A Felirat osztályban az osztály és a metódusok elé /\*\*-gal kezdődően írtuk be ezeket a szövegeket. A metódusoknál jelöltük a paraméterek és a visszaadott érték jelentését is.

Ezután a dokumentáció elkészítéséhez rá kell kattintani a jobb gombbal a projekt nevére, majd a menüből a Generate Javadoc parancsot kell választani:



Próbáld ki! Figyeld meg, hogy a dokumentáció HTML formátumban készül, és automatikusan megnyílik a böngészőben!

Nézd meg a Felirat osztály dokumentációját! (A txt2srt még üres.) Szükség estén javítsd vagy pótold a hiányzó részeket, majd generáld újra a dokumentációt!

## Kérdések

1. Milyen szófajt szoktunk használni az osztályok nevére? És a metódusokéra?
2. Hogyan lehet egy másik osztályból módosítani a feliratok időzítését vagy szövegét? Miért?

# 24. Osztályok 4.

## Feliratok

Ebben a leckében befejezzük a felirat átalakító programot.

Emlékeztető: Ebben a feladatban egy videóhoz SRT formátumú, angol nyelvű feliratfájlt kell készítened. A feliratok és azok időzítései rendelkezésre állnak a *feliratok.txt* állományban, de a formátumuk nem megfelelő.

A *feliratok.txt* állományban 2-2 soronként ismétlődve egy felirat időzítése és az ehhez tartozó felirat található:

00:01 - 00:03  
So phase two - tank creation.

A példában a felirat a videó lejátszásakor az első másodpercnél jelenik meg ( 00:01 ), a harmadik másodpercben tűnik el ( 00:03 ), és a megjelenített felirat szövege a „So phase two - tank creation”.

(Forrás: <https://unity3d.com/learn/tutorials/projects/tanks-tutorial>)

Nyisd meg a múltkor elkészített projektet, és nézd át az eddigi kódokat!

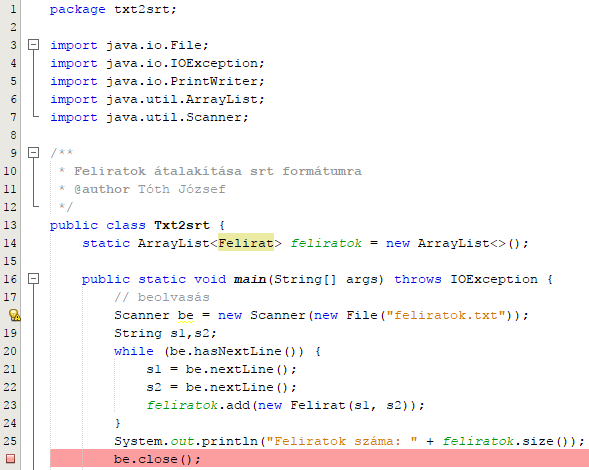
Nézd meg a múltkor elkészített dokumentációt! Melyik mappában van?

## Beolvasás

Most már átválthatunk a Txt2srt osztályra, és elkezdhetjük az átalakítást.

Első feladat az adatok beolvasása. Az adatokat egy feliratok nevű listába olvassuk be, amelynek elemei Felirat típusúak lesznek. Utána kiíratjuk a lista méretét.

Nézzük a beolvasás kódját:



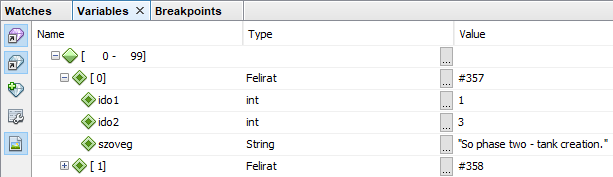
Az importálásokat automatikusan is el lehet végeztetni a Ctrl+Shit+I megnyomásával.

A kódban van egy rövidítés: A 18. sorban az új File objektumot közvetlenül a Scanner konstruktorának adtuk meg, így eggyel kevesebb változóra van szükség.

Feltételezzük, hogy a fájlban párosával vannak a sorok.

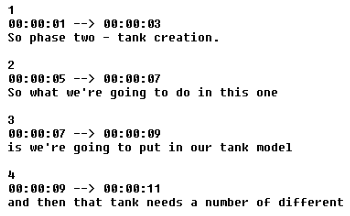
Próbáld ki! Hány feliratot olvasott be? Írd ide:

Helyezz el egy töréspontot a 26. sorban! Indítsd el a debuggert (Ctrl+F5), és ellenőrizd a feliratok lista tartalmát! Utána töröld a töréspontot!



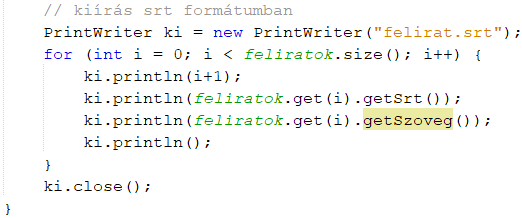
## Srt fájl elkészítése

Az srt fájl formátuma:



Egy ciklussal végigmegyünk a listán, és minden elemnél kiíratjuk a ciklusváltozó értékét, az átalakított időzítést és a (változatlan) szöveget.

Egészítsd ki a főprogramot így:



Ha a szöveges fájlhoz nem hozzáfűzni akarunk, hanem új tartalmat írni bele, akkor a PrintWriter objektum közvetlenül a fájlnévvel is használható. (Nem kell a FileWriter objektum.)

Miért i+1-et íratunk ki a felirat sorszámaként?

Próbáld ki, és ellenőrizd a *feliratok.srt* fájlt a projekt mappájában!

## Feladatok

1. Egészítsd ki a programot úgy, hogy a beolvasás után írja ki a leghosszabb feliratot, és hogy hány karakterből áll!
2. Kiírás előtt tolja el a program az időzítéseket 1 másodperccel!

# 25. Osztályok 5.

## Kéktúra

Az Országos Kéktúra Magyarország északi részén végighaladó folyamatos, jelzett turistaút. A Kéktúrának a Balaton-felvidéken is több, rövidebb idő alatt bejárható túrája van. Egy ilyen túra adatait kell feldolgoznia ebben a feladatban.

A *kektura*.*csv* állomány első sorában a túra kezdetének tengerszint feletti magassága található méterben megadva. A további sorok a túra egy-egy szakaszához tartozó adatokat tárolják. Az adatokat pontosvessző választja el egymástól. Például:

Hidegkuti major;Leteres a Sztupahoz;3,903;153;53;n

A sorokban lévő adatok rendre a következők:

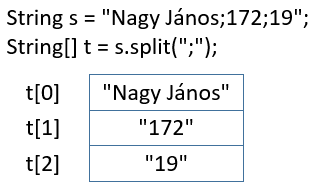
* A túraszakasz kiindulópontjának a neve.   
  Például: „Hidegkuti major”
* A túraszakasz végpontjának a neve.   
  Például: „Leteres a Sztupahoz”
* A túraszakasz hossza [km]. Például: „3,903”
* A túraszakaszon mért emelkedések összege [m].   
  Például: „153”
* A túraszakaszon mért lejtések összege [m]: Például: „53”
* A szakasz végpontja pecsételőhely-e: „i” vagy „n”

Nézd meg a *kektura.csv* fájl tartalmát!

## String szétvágása

A fájlban egy soron belül sok adat van, pontosvesszővel elválasztva. Ezt úgy lehet könnyen feldolgozni, hogy a sorokat szétvágjuk a pontosvesszőknél, és a részeket elhelyezzük egy string tömbben.

Erre szolgál a stringek split() metódusa. Paraméterként azt a karaktert vagy azokat a karaktereket kell megadni, amelyeknél a vágásokat el kell végezni. Például:



Arra vigyázz, hogy a tömb elemei stringek, akkor is, ha számjegyeket tartalmaznak!

## Osztályok létrehozása

Először meg kell terveznünk, hogy milyen adatszerkezetben tároljuk az adatokat.

Minden túraszakasz adatait egy-egy objektumban fogjuk tárolni, és ezekből hozunk majd létre egy listát. Az objektum típusa Szakasz lesz. A Szakasz osztály hat adattagot fog tartalmazni a fenti felsorolásnak megfelelően.

A Szakasz osztályt külön fájlba tesszük:

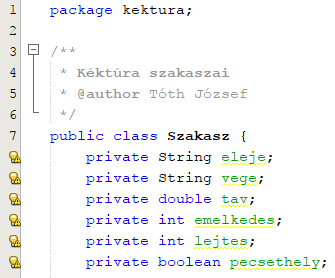
1. Először kezdj egy új projektet *kektura* néven, a szokott módon!
2. A Projects ablakban kattints a jobb gombbal a *kektura* csomagra, majd a menüből választd a New, Java Class... parancsot!
3. Az osztály neve legyen Szakasz (nagy S-sel!), majd kattints a Finish gombra!

Így most már két osztályból áll a programunk:

* A Szakasz osztályban definiáljuk az egyes szakaszok kezeléséhez szükséges adattagokat és metódusokat.
* A Kektura osztályban van a main() metódus, a program indításakor az fog elindulni.

## A Szakasz osztály

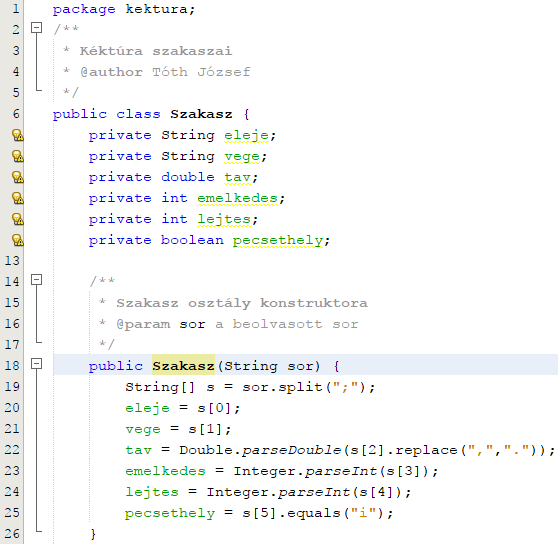
A Szakasz osztályban először megadjuk az adattagokat. Írd be a következőket: (A szerző nevéhez írd a saját nevedet!)



Mivel a Szakasz osztály példányait használjuk majd, itt nem kell a static szó sem az adattagok, sem a metódusok elé.

A privát adattagokat publikus metódusokkal fogjuk kezelni.

Egészítsd ki a Szakasz osztályt egy olyan konstruktorral, amely egy stringet kap, és ebből beállítja az adattagok kezdőértékeit:

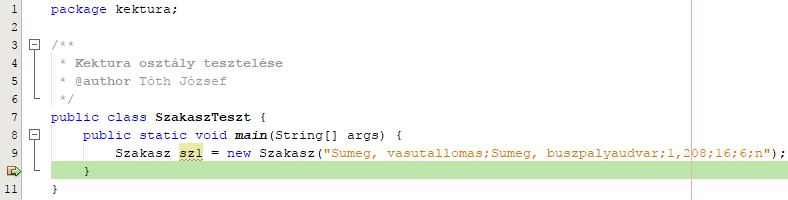


Figyeld meg a split() metódus használatát és a típusátalakításokat!

A 22. sorban azért helyettesítettük a vesszőket pontokkal, mert a parseDouble() metódus csak tizedespontot fogad el (tizedesvesszőt nem).

## Tesztelés

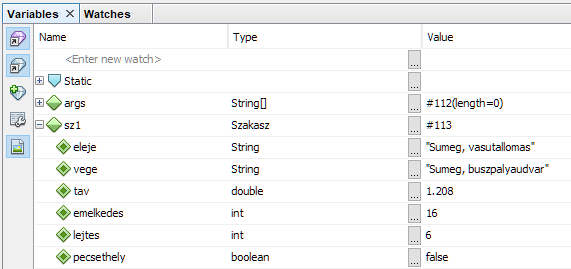
Készíts egy main metódust a Szakasz osztályban teszteléséhez:



A paraméter szövegét átmásolhatod a *kektura.txt* fájl első sorából.

Helyezz el egy töréspontot a main metódus végén, majd indítsd el a debuggert (Ctrl+Shift+F5, vagy jobb kattintás és Debug File)!

Ellenőrizd az sz1 objektum adattagjait, majd kapcsold ki a töréspontot!

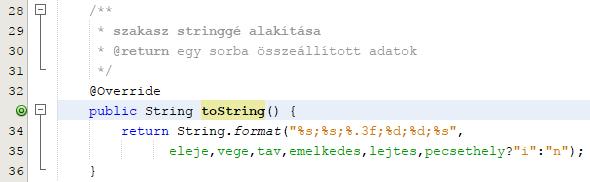


## Objektum kiíratása

Minden objektumnak van egy toString() metódusa. Ez azonban csak az objektum nevét és egy címet ad meg, így nem sok mindenre használható.

Mivel a programban több helyen is ki kell majd íratnunk egy-egy szakasz adatait, célszerű ezt az (Object osztálytól) örökölt metódust felülírni egy használhatóval.

Egészítsd ki a Szakasz osztályt a toString() metódussal:

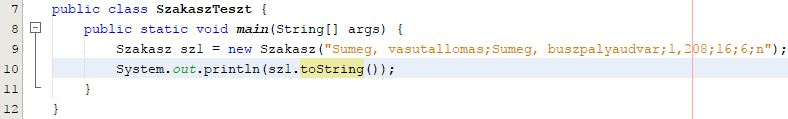


Az @override annotáció jelzi, hogy ez egy örökölt metódus átírása.

És van a 35. sorban még egy újdonság: az egysoros if utasítás.

Ennek a kifejezésnek:   
pecsethely?"i":"n"  
az értéke a pecsethely változó értékétől függ. Ha a pecsethely értéke igaz, akkor az eredmény "i", egyébként pedig "n".

Egészítsd ki a tesztet az alábbiak szerint, majd próbáld ki:



Megjegyzés: a println metódusban elég lenne az sz1-et megadni, mert a println metódus úgyis meghívja a toString metódust.

Tehár a kiírás így is jó:



A következő leckében folytatjuk.

## Kérdések, feladatok

1. Ha az x változó értéke 8, akkor mi lesz az s változó értéke?  
   String s = (x % 2 == 0 ? "páros" : "páratlan")
2. Írd le ugyanezt if utasítással:
3. Ha m egy több szóból álló string, a szavak között egy-egy szóközzel, akkor mit ad meg az alábbi kifejezés?  
   m.split(" ").length

# 26. Osztályok 6.

## Kéktúra

Ebben a leckében folytatjuk a Kéktúrával kapcsolatos feladat megoldását.

Emlékeztető:

A *kektura*.*csv* állomány első sorában a túra kezdetének tengerszint feletti magassága található méterben megadva. A további sorok a túra egy-egy szakaszához tartozó adatokat tárolják. Az adatokat pontosvessző választja el egymástól. Például:

Hidegkuti major;Leteres a Sztupahoz;3,903;153;53;n

A sorokban lévő adatok rendre a következők:

* A túraszakasz kiindulópontjának a neve.   
  Például: „Hidegkuti major”
* A túraszakasz végpontjának a neve.   
  Például: „Leteres a Sztupahoz”
* A túraszakasz hossza [km]. Például: „3,903”
* A túraszakaszon mért emelkedések összege [m].   
  Például: „153”
* A túraszakaszon mért lejtések összege [m]: Például: „53”
* A szakasz végpontja pecsételőhely-e: „i” vagy „n”

Nézd meg a *kektura.csv* fájl tartalmát!

A következőkben meghatározzuk:

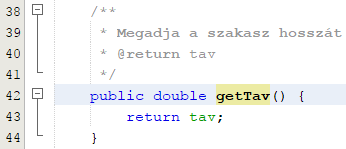
* a túra teljes hosszát,
* a három leghosszabb szakasz hosszait, és
* a legrövidebb szakasz hosszát.

Közben megismerkedünk a lambda kifejezésekkel és megtanuljuk sorba rendezni az objektumokat.

## Teljes hossz kiszámítása

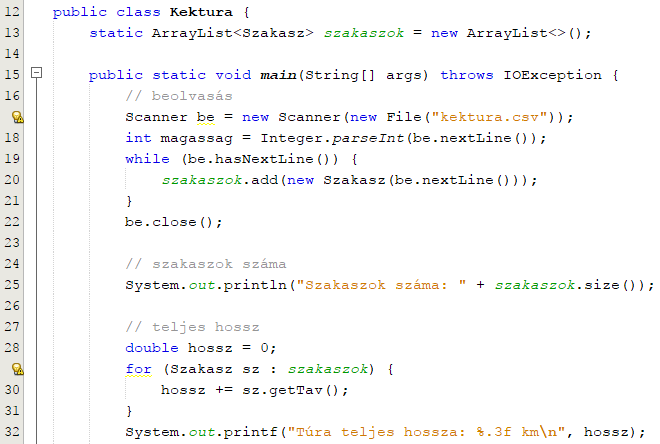
Következő feladatunk a túra teljes hosszának meghatározása. Ehhez végig kell menni az összes szakaszon, és össze kell adni az egyes távokat.

Az első lépés, hogy a Szakasz osztályban készítünk egy publikus metódust, amellyel le lehet kérdezni egy szakasz hosszát. Egészítsd ki a Szakasz osztályt így:



Ez a metódus annyira egyszerű, hogy nem fogjuk tesztelni.

A főprogramban létrehozzuk a listát, beolvassuk az adatokat, kiírjuk a szakaszok számát, majd kiszámítjuk a teljes hosszt:



Figyeld meg a for-each ciklust és a printf() használatát!

Próbáld ki! Mennyi a teljes hossz? Írd ide:

## Lambda kifejezések

A lambda kifejezések a Java 8-as változatától kezdve használhatók, és sokszor lényegesen lerövidítik a szükséges kódot.

Egy lambda kifejezés tulajdonképpen egy egyszerűsített metódus.

Vegyük például a következő metódust:

int plusz(int a, int b) {  
 return a + b;  
}

Ez lambda kifejezéssel így írható:

(a, b) -> a + b

Figyeld meg, hogy:

* ennek a metódusnak nincs neve,
* a paramétereket zárójelek között kell megadni, utána nyíllal,
* a visszaadott értéknek és a paramétereknek nincs megadva a típusa.

A lambda kifejezéseket akkor fogjuk használni, amikor paraméterként kell megadni egy metódust, így nem lesz szükség a névre.

A típusokat a fordító fogja kitalálni a helyzetnek megfelelően (nem csak egészekkel működik!)

a és b helyett tetszőleges neveket használhatunk.

## Objektumok sorba rendezése

Már tanultuk, hogy egy lista objektumait a Collections.sort() metódussal tudjuk sorba rendezni. (Tömbök esetén pedig az Arrays.sort() metódust használjuk.)

Integer, Double vagy String típusú objektumok esetén egyértelmű, hogyan kell rendezni.

Más objektumok, például Szakasz típusú objektumok esetén viszont nem tudja a fordító, hogyan hasonlítsa össze őket. Ilyenkor a sort metódusnak megadunk egy lambda kifejezést, amely meghatározza az összehasonlítást.

A lambda kifejezésben a és b két objektum, az eredmény pedig:

* negatív, ha a<b,
* 0, ha a=b,
* pozitív, ha a>b.

A stringek CompareTo() metódusa is pontosan így működik. Ezért két szöveges adattag összehasonlítását így adjuk meg (itt feltételezzük, hogy a getEleje() metódus az eleje adattag értékét adja meg):

(a, b) -> a.getEleje().compareTo(b.getEleje())

Ez alapján a sort() metódussal a szakaszok induló állomása szerint növekvő sorrendbe rendeznénk a listát. Hogyan lehetne csökkenő sorrendet megadni?

Egész számok esetén alkalmazhatjuk az Integer.compare metódust. Ha például a szintvaltozas metódus azt adja meg, hogy mennyi az emelkedés és a lejtés különbsége az adott szakaszon, akkor így rendezhetünk e szerint:

(a, b) -> Integer.compare(a.szintvaltozas(), b.szintvaltozas())

Valós számok esetén használhatjuk a Double.compare metódust. Ha például a távok szerint növekvő sorrendbe szeretnénk rendezni, akkor ezt a lambda kifejezést adjuk meg a sort() metódusnak:

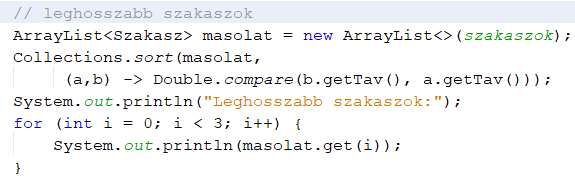
(a, b) -> Double.compare(a.getTav(), b.getTav())

Hogyan lehet csökkenő sorrendet megadni?

## Leghosszabb szakaszok

Most az a feladat, hogy írassuk ki a három leghosszabb szakasz adatait. Ezt a következő lépésekkel végezzük el:

1. Készítünk egy másolatot a listáról, mert később az eredeti sorrendben is szükségünk lesz rá.
2. A másolatot csökkenő sorrendbe rendezzük.
3. Kiíratjuk a csökkenő sorrendbe rendezett lista első három elemét.

Egészítsd ki a főprogramot az alábbiak szerint:  


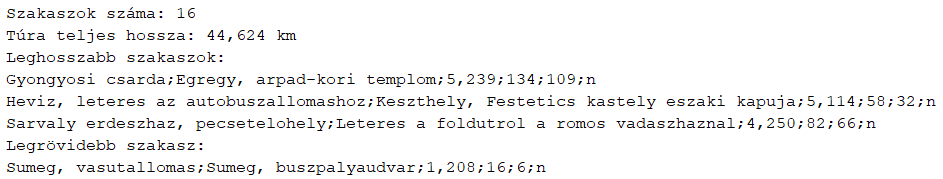
Figyeld meg, hogyan készítettünk másolatot a listáról, és hogyan alkalmaztuk a lambda kifejezést!

Próbáld ki a programot!

A következő leckében befejezzük a feladat megoldását.

## Önálló feladat

Írasd ki a legrövidebb szakasz adatait is! Minta:



# 27. Osztályok 7.

## Kéktúra

Ebben a leckében befejezzük a Kéktúrával kapcsolatos feladat megoldását.

Emlékeztetó:

A *kektura*.*csv* állomány első sorában a túra kezdetének tengerszint feletti magassága található méterben megadva. A további sorok a túra egy-egy szakaszához tartozó adatokat tárolják. Az adatokat pontosvessző választja el egymástól. Például:

Hidegkuti major;Leteres a Sztupahoz;3,903;153;53;n

A sorokban lévő adatok rendre a következők:

* A túraszakasz kiindulópontjának a neve.   
  Például: „Hidegkuti major”
* A túraszakasz végpontjának a neve.   
  Például: „Leteres a Sztupahoz”
* A túraszakasz hossza [km]. Például: „3,903”
* A túraszakaszon mért emelkedések összege [m].   
  Például: „153”
* A túraszakaszon mért lejtések összege [m]: Például: „53”
* A szakasz végpontja pecsételőhely-e: „i” vagy „n”

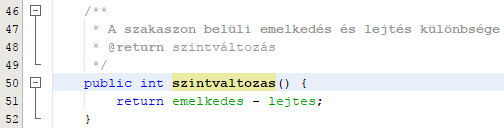
Nézd meg a *kektura.csv* fájl tartalmát!

Ma először megkeressük a legmagasabb végpontot, azután kiíratjuk a hiányos állomásneveket, amelyeket végül ki is javítunk.

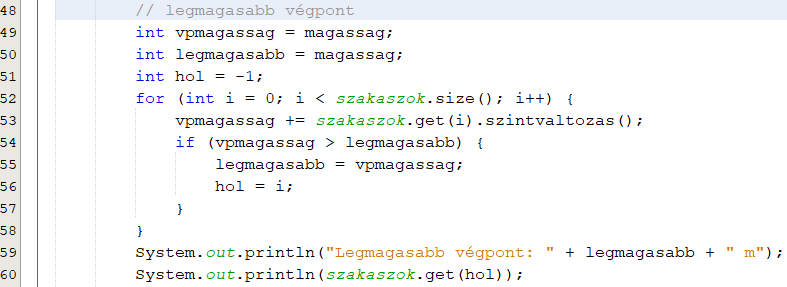
## Legmagasabb végpont

Ismerjük a túra kiindulópontjának tengerszint feletti magasságát és az egyes szakaszokon mért emelkedések és lejtések összegeit. Az adatok ismeretében keressük meg a túra legmagasabban fekvő végpontját és határozzuk meg a végpont tengerszint feletti magasságát! Feltételezzük, hogy csak egy ilyen végpont van.

Az első lépés, hogy a Szakasz osztályban készítünk egy publikus metódust, amellyel le lehet kérdezni egy szakaszon belül a szintváltozást. Egészítsd ki a Szakasz osztályt így:



Ez a metódus is annyira egyszerű, hogy nem fogjuk tesztelni, hanem egyből alkalmazzuk a főprogram végén:



Egy ciklussal végigmegyünk a túra szakaszain (az eredeti sorrendben!)

A vpmagassag változó értéke az i-dik szakasz végpontjának magassága.

A legmagasabb változó tárolja az eddig mért legmagasabb értéket. Ez a fájl elejéről beolvasott magasságról indul.

A hol változó azt adja meg, hogy hányadik szakasz eddig a legmagasabb.

Próbáld ki a programot! Mekkora a legmagasabb végpont tengerszint feletti magassága? Írd ide:

## Hiányos állomásnevek

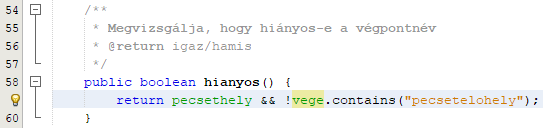
Készíts a Szakasz osztályban logikai értéket visszaadó metódust hianyos azonosítóval, amelynek segítségével minősíteni tudod a túraszakaszok végpontjainak a nevét!

Hiányos állomásneveknek minősítjük azokat a végpontneveket, amelyek pecsételőhelyek, de a „pecsetelohely” karakterlánc nem található meg a nevükben. Ebben az esetben logikai igaz értéket adjon vissza a metódus, egyébként pedig hamisat!

A megoldáshoz felhasználjuk a stringek contains() metódusát, amely megadja, hogy a paraméterként kapott string szerepel-e abban a stringben, amelyikre meghívtuk.

Például a következő kifejezés:  
s.contains("palyaudvar")  
értéke true, ha az s string tartalmazza a palyaudvar szót, egyébként pedig hamis.

Írd be a Szakasz osztály végére a metódust:

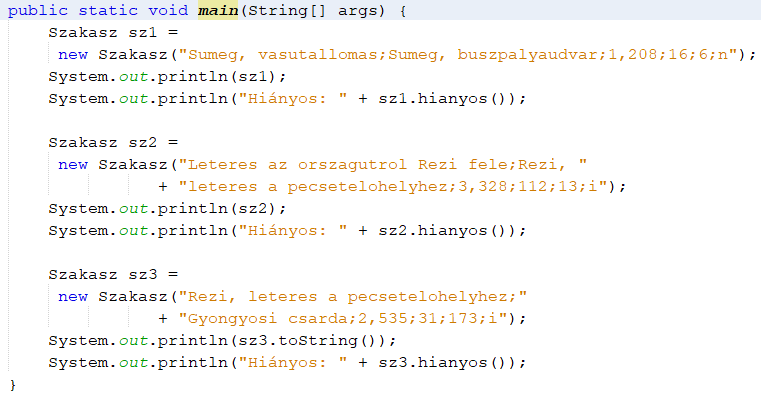


Mivel ez nem olyan egyszerű, először teszteljük a működését. Mit gondolsz, milyen teszteseteket kell vizsgálnunk?

A vizsgálandó eseteket a következő táblázat mutatja:

|  |  |  |
| --- | --- | --- |
| Pecsételőhely | Tartalmazza a név | Várt eredmény (hiányos-e) |
| nem | nem | hamis |
| igen | igen | hamis |
| igen | nem | igaz |

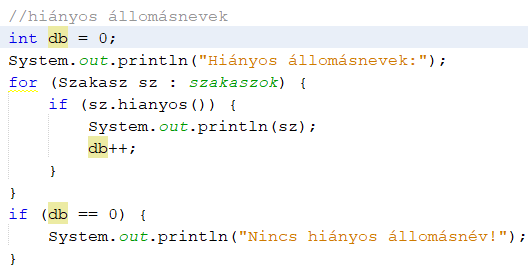
Keressünk a három lehetséges esetnek megfelelő sorokat, és teszteljük ezek segítségével a metódust:



A szakaszok létrehozását természetesen egy-egy sorba is írhatod.

Ez alapján írassuk ki a képernyőre a hiányos állomásneveket! Ha nincs hiányos állomásnév az adatokban, akkor a „Nincs hiányos állomásnév!” felirat jelenjen meg!

Gondold végig, és írd be a megoldást a főprogram végére:

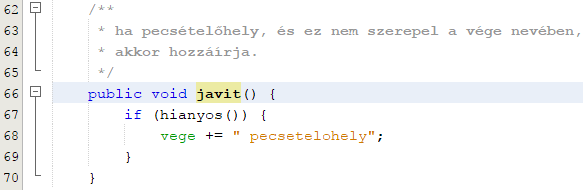


Próbáld ki! Hány hiányos állomásnév van? Írd ide:

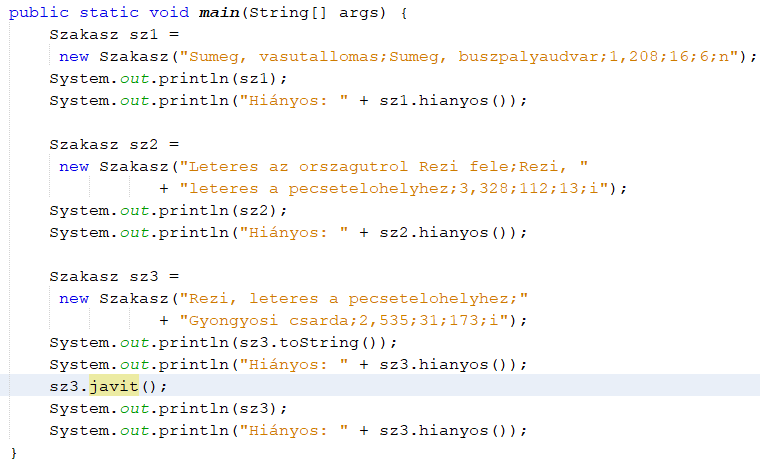
## Állomásnevek javítása

Készíts *kektura2.csv* néven szöveges állományt, amelynek szerkezete megegyezik a *kektura.csv* állományéval! A kimeneti fájl első sora a kiindulópont tengerszint feletti magasságát tartalmazza! A további sorokban a túra szakaszainak adatait írja ki! Azoknál a pecsételőhelyeknél, ahol nem található meg a végpont nevében a „pecsetelohely” karaktersorozat, ott kerüljön be a végpont nevének a végére egy szóközzel elválasztva a „pecsetelohely” szó!

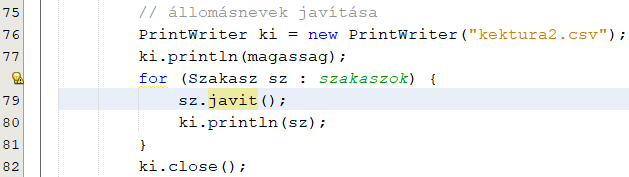
Először készítsd el a javit() metódust a Szakasz osztályban:



Utána egészítsd ki a tesztet, és próbáld ki!



A javit() metódus segítségével már el tudjuk készíteni a főprogrammal a javított fájlt:



Próbáld ki, és ellenőrizd a *kektura2.csv* fájlt (a projekt mappájában)!

# 28. Osztályok 8.

Egy iskolában *Ki mit tud* versenyt rendeznek. Minden versenyző egy 6 tagú zsűri előtt lép fel, és a zsűritagok 1-től 10-ig pontozzák a teljesítményét. Az így elért pontszám összegéből a legmagasabb és a legalacsonyabb pontszámot kivonják, és ez lesz végül a versenyző összesített pontszáma.

Ebben a feladatban a selejtező lebonyolításához kell elkészítenie egy alkalmazást.

A megoldás során vegye figyelembe a következőket:

* *A képernyőre írást igénylő részfeladatok eredményének megjelenítése előtt írja a képernyőre a feladat sorszámát (például:3. feladat:)!*
* *A program megírásakor a fájlban lévő adatok helyes szerkezetét nem kell ellenőriznie, feltételezheti, hogy a rendelkezésre álló adatok a leírtaknak megfelelnek.*
* *A megoldását úgy készítse el, hogy az azonos szerkezetű, de tetszőleges input adatok mellett is helyes eredményt adjon!*

A selejtezo.txt állomány soronként tartalmazza a versenyen részt vevő játékosok nevét és a pontszámaikat. A nevek a pontszámoktól pontosvessző karakterrel vannak elválasztva, a pontszámok között szóköz karakter található. Például:

Kovács Adél;6 10 9 8 7 6

A sorokban lévő adatok rendre a következők:

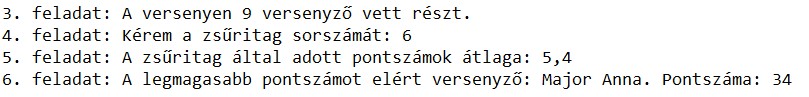
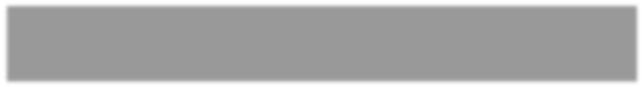
* A versenyző neve. Például: „Kovács Adél”
* A zsűritagok pontszámai. Például: 6 10 9 8 7 6

(Adél első zsűritagtól kapott pontszáma 6, a második zsűritagtól kapott pontszáma 10, és a hatodik zsűritagtól szintén 6 pontot kapott.)

Készítsen ***konzolos*** alkalmazást a következő feladatok megoldására, a projektet KiMitTud néven mentse el!

1. Olvassa be a selejtezo.txt állományban lévő adatokat, és tárolja el egy olyan adatszerkezetben, ami a további feladatok megoldására alkalmas!
2. Határozza meg és írja ki a képernyőre a minta szerint, hogy a selejtezőn hány versenyző vett részt!
3. Kérje be egy zsűritag sorszámát! Feltételezheti, hogy az input adat 1 és 6 közötti pozitív egész szám.
4. Határozza meg és írja ki a minta szerint az előző feladatban bekért sorszámú zsűritag által adott pontok átlagát! Az eredmény 1 tizedesjegyre kerekítve jelenjen meg!
5. Határozza meg, és írja ki a minta szerint, hogy ki volt a legmagasabb összesített pontszámot elérő versenyző, és mennyi volt az elért pontszáma!
6. Írassa ki a legmagasabb pontszámot elért versenyző nevét és pontszámát a minta szerint!

MINTA a ***konzolos*** alkalmazáshoz:



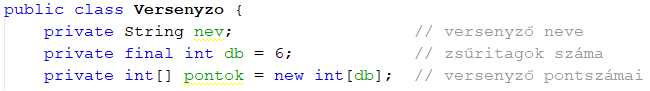
## 1. feladat: Új projekt

Kezdj egy új projektet kimittud néven!

## 2. feladat: beolvasás, tárolás

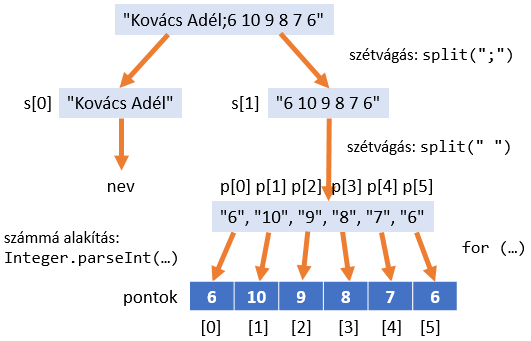
Egy versenyző adatainak tárolásához létrehozunk egy osztályt, amelyben a nevét és a pontszámait tároljuk, és a versenyzővel kapcsolatos műveleteket végezzük. (jobb kattintás a csomag nevére, New, Java Class). Ezt az osztályt Versenyzo-nak nevezzük el.

A Versenyzo osztályban tárolni kell a versenyző nevét (String) és pontszámait (egészekből álló tömb). A programot úgy készítjük el, hogy más zsűriszám esetén is működjön, ezért létrehozunk egy db nevű állandót, amely a zsűritagok számát adja meg.



A fájlból történő beolvasáskor a beolvasott sorokból kell létrehozni egy-egy új versenyzőt.

Gondoljuk végig, hogyan történik a sor átalakítása:



A sort először a pontosvesszőnél vágjuk szét, és az elemeket elhelyezzük az s nevű String tömbben. A s[0]-ból kiolvassuk a versenyző nevét.

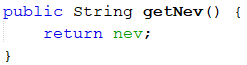
Ezután az s[1]-et még egyszer szétvágjuk, most a szóközöknél, és a p tömbbe tesszük az eredményt.

Végül a p tömb elemeit egész számokká alakítjuk az Integer.parseInt metódus segítségével, és elhelyezzük a pontok tömbben.

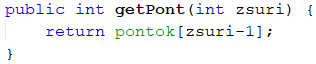
Ezt a következő konstruktor végzi majd el:



A név kiolvasásához készítünk egy getter metódust:

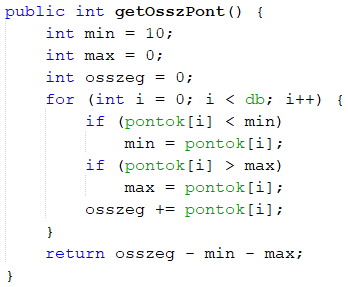


Az 5-ös feladathoz készítünk egy metódust, amely egy megadott zsűritag által adott pontszámot adja vissza:



A -1 azért kell, mert a zsűritagokat 1-től kezdve számozzuk, a tömbelemeket viszont 0-tól kezdve.

Végül kell egy olyan metódus, amely kiszámolja a versenyző összpontszámát a megadott szabályok szerint:



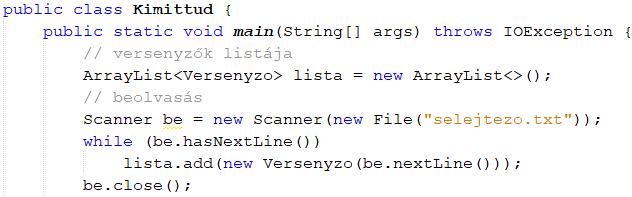
Ez a metódus végig megy a pontszámokon, közben összeadja őket és megkeresi a legnagyobbat és a legkisebbet. Végül a pontszámok összegéből kivonja a legnagyobbat és a legkisebbet.

Ezzel a Versenyzo osztály el is készült.

Mivel nem tudjuk előre, hogy hány versenyző van, a főprogramban a versenyzők adatait egy listában tároljuk. A lista elemei a Versenyzo osztály példányai lesznek. Mivel a main metódusban fogunk mindent megoldani, a listát is a main metódusban hozzuk létre.

A lista létrehozása után beolvassuk az adatokat a *selejtezo.txt* fájlból. Ne felejtsd el ezt a fájlt bemásolni a projekt mappájába!

Írd ezt az Kimittud osztály main metódusába:

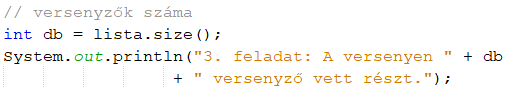


A feladat nem kért hibakezelést, ezért csak jelezzük, hogy a main metódusban kivétel keletkezhet.

Futtasd a programot! Egyelőre csak annyit kell látnod, hogy hiba nélkül lefut.

## 3. feladat: versenyzők száma

A versenyzők száma megegyezik a Versenyzok lista hosszával. Folytasd így a main metódust:

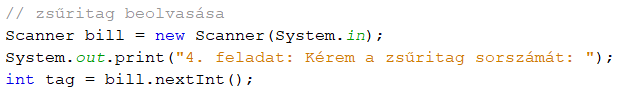


A versenyzők számát azért tároljuk a db változóban, mert az átlagszámításnál is szükség lesz rá.

Próbáld ki és ellenőrizd a minta alapján!

## 4. feladat: zsüritag beolvasása

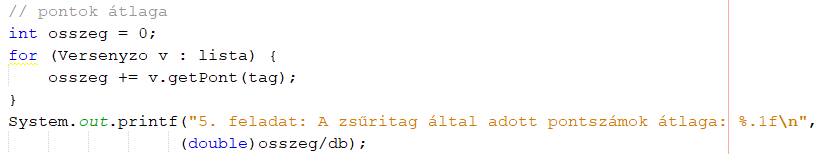
Ezután beolvassuk egy zsűritag sorszámát ellenőrzés nélkül:



Próbáld ki!

## 5. feladat: zsűritag pontjainak átlaga

Az átlag számításához először összegezzük a zsűritag által adott pontszámokat, majd kiíratjuk az osszeg és a db hányadosát (feltételezzük, hogy a pontok száma nem 0):

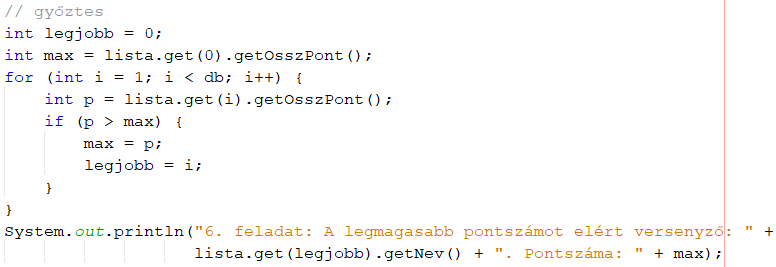


A valós osztáshoz az osszeg-et double típusúra alakítjuk. Az egy tizedesjeggyel történő megjelenítést a %.1f kód jelzi a printf metódusban.

Próbáld ki és ellenőrizd a minta alapján!

## 6. feladat: győztes

A vetélkedő győztesének meghatározásához megkeressük a legnagyobb összpontszámot:



Próbáld ki és ellenőrizd a minta alapján!

# 29. Osztályok 9.

## Színház

A Fregoli Színházban a jegyeladásokat elektronikusan rögzítik. A színházban 15 sor, és soronként 20 szék van. A sorokat 1-től 15-ig számozzák, a sorokon belül pedig a székeket 1-től 20-ig.

Egy előadásra a pillanatnyilag eladott jegyek eloszlását a *foglaltsag.txt* nevű szöveges állomány tartalmazza, amelyben „x” jelzi a foglalt és „o” a szabad helyeket.

Például:

ooxxxoxoxoxoxxxooxxx  
xxxxxxxxxxxxxxxxxxxx  
oxxxxxoooxxxxxxxxxxo  
…

Az első sor 1-2. széke például még szabad, míg a 2. sorba az összes jegyet eladták.

A jegyek ára nem egyforma, összege a helytől függően ötféle lehet. Azt, hogy az adott szék az öt közül melyik árkategóriába tartozik, a *kategoria.txt* nevű fájl tartalmazza az alábbi formában:

Például:

33222111111111122233  
43322221111112222334  
44433322222222333444  
…

A példa szerint az 1. sor 2. széke a 3. kategóriába, a 2. sor 1. széke a 4. kategóriába esik.

Nézd meg a szöveges fájlok tartalmát!

Hogyan tudnád számítógép nélkül elvégezni a következő műveleteket?

* adott szék (sor, szám) foglaltságának vizsgálata,
* összes eladott jegy megszámlálása,
* bevétel kiszámítása,
* egyes üres helyek megszámlálása (szomszédjai foglaltak).

## Osztályok létrehozása

Először meg kell terveznünk, hogy milyen adatszerkezetben tároljuk az adatokat.

Minden széksor adatait egy-egy objektumban fogjuk tárolni, és ezekből hozunk majd létre egy tömböt. Azért nem listát használunk, mert a méretek rögzítettek.

Az objektum típusa Szeksor lesz. A Szeksor osztály két mezőt fog tartalmazni: a foglalt és a kategoria mezőket, amelyek egy-egy sornak felelnek meg a szöveges fájlokban, és egy-egy stringben tárolhatók.

A Szeksor osztályt külön fájlba tesszük:

1. Először kezdj egy új projektet *szinhaz* néven, a szokott módon!
2. A Projects ablakban kattints a jobb gombbal a *szinhaz* csomagra, majd a menüből válaszd a New, Java Class... parancsot!
3. Az osztály neve legyen Szeksor (nagy S-sel!), majd kattints a Finish gombra!

Így most már két osztályból áll a programunk:

* A Szeksor osztályban definiáljuk az egyes széksorok kezeléséhez szükséges mezőket és metódusokat.
* A Szinhaz osztályban van a main() metódus, a program indításakor az fog elindulni.

## A Szeksor osztály

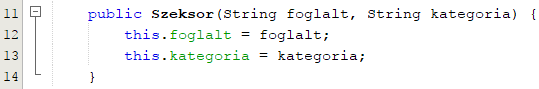
A Szeksor osztályban először megadjuk az adattagokat. Írd be a következőket: (A szerző nevéhez írd a saját nevedet!)



Mivel a Szeksor osztály példányait használjuk majd, itt nem kell a static szó sem az adattagok, sem a metódusok elé.

A privát adattagokat publikus metódusokkal fogjuk kezelni.

Egészítsd ki a Szeksor osztályt a konstruktorral, amely két stringet kap, és ebből beállítja az adattagok kezdőértékeit:  
(A NetBeans automatikusan elkészíti a konstruktort, ha megnyomod az Alt+Insert billentyűket, kiválasztod a Constructor parancsot, majd bepipálod a mezőket.)



Mit jelent a this szó, és miért használtuk itt?

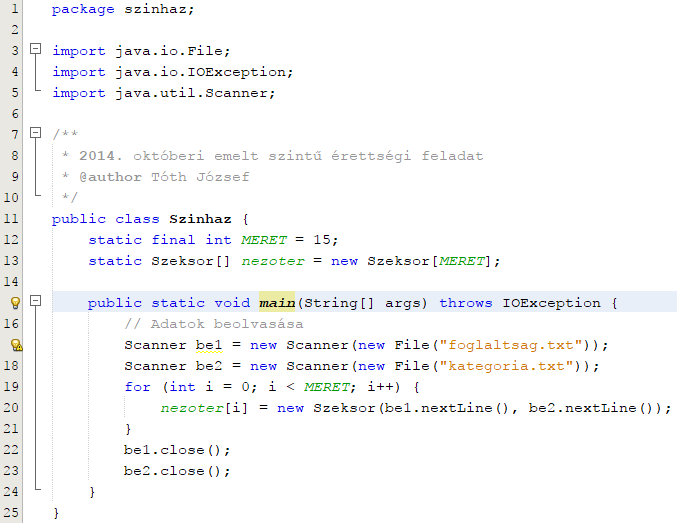
## Adatok beolvasása

Ezek után megírhatjuk a főprogramnak azt a részét, amely beolvassa az adatokat a fájlokból.

A színházban 15 széksor van. Ezt a méretet egy állandóban rögzítjük.

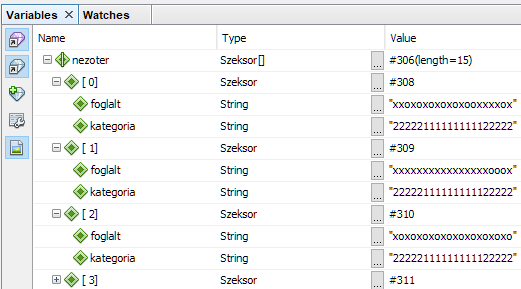
Utána létrehozunk egy 15 elemű, széksorokból álló tömböt.

Végül beolvassuk a 15-15 sor adatot mindkét fájlból:



Ne felejtsd el a kipróbálás előtt a szöveges fájlokat a projekt mappájába másolni!

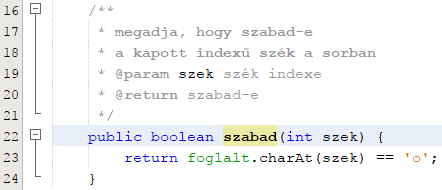
Helyezz el egy töréspontot a fájlok bezárásához, és ellenőrizd a beolvasott adatokat!



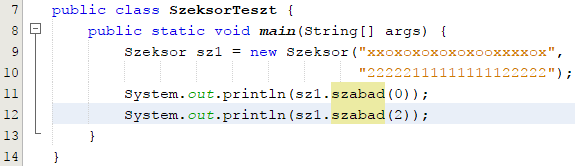
## Foglaltság lekérdezése

Ezután be kell kérni a felhasználótól egy sor és egy székszámot, és kiírni, hogy a megadott ülés szabad-e.

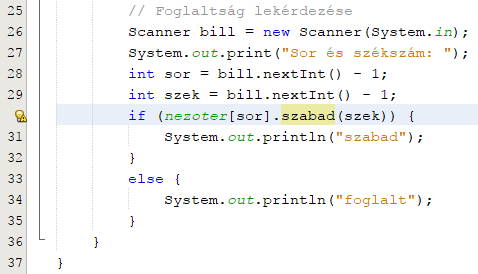
Ehhez először a Szeksor osztályban készíts egy metódust, amely megadja, hogy abban a sorban a kapott indexű szék szabad-e:



Teszteld ezt a Szeksor osztályban egy main metódussal:



Végül egészítsd ki a Szinhaz osztály főprogramját:



Próbáld ki egy foglalt és egy szabad székkel is!

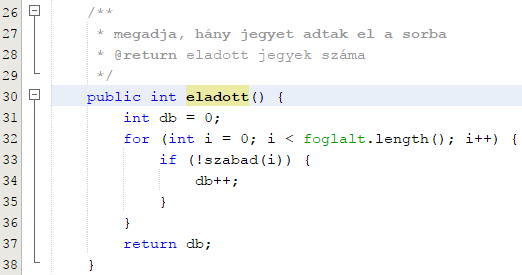




## Eladott jegyek

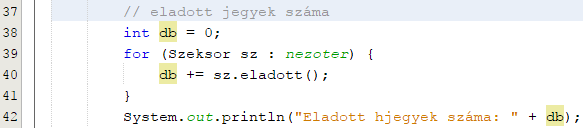
Ezután azt kell meghatároznod, hogy hány jegyet adtak el.

Készíts egy metódust a Szeksor osztályba, amely megadja, hogy az adott sorban hány jegyet adtak el:



Teszteld Szeksor osztály main metódusában az eladott metódus működését az sz1 változóval önállóan! Mennyi lett az eredmény?

Ezután egészítsd ki a Szinhaz osztály főprogramját az eladásokat összegző résszel:



Próbáld ki! Hány jegyet adtak el összesen? Írd ide:

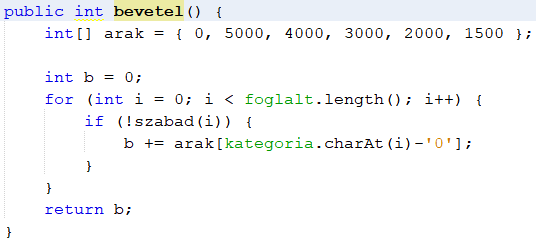
## Bevétel

A jegyek árát kategóriánként a következő táblázat tartalmazza:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| árkategória | 1 | 2 | 3 | 4 | 5 |
| ár (Ft) | 5000 | 4000 | 3000 | 2000 | 1500 |

Mennyi lenne a színház bevétele a pillanatnyilag eladott jegyek alapján? Írasd ki az eredményt a képernyőre!

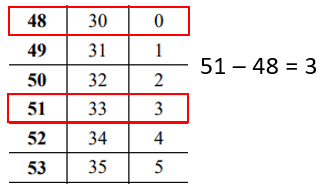
Először egy széksorra határozzuk meg a bevételt. Írd ezt a Szeksor osztályba:



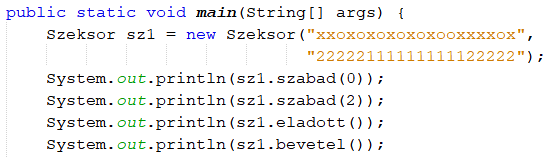
Az arak tömb tartalmazza az árakat. A kategória az ár indexét határozza meg. 0-ás kategória nincs.

A metódus végigmegy a foglalt string karakterein, és a foglalt székek árát összegzi.

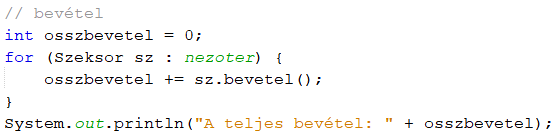
A kategoria.charAt(i)-'0' kifejezéssel az i-dik karakter értékét határozzuk meg. utána az árakat tartalmazó tömbből az értéknek megfelelő elemet vesszük.



Teszteld a metódust:





Egészítsd ki a szinhaz osztály main metódusát az alábbiakkal:  


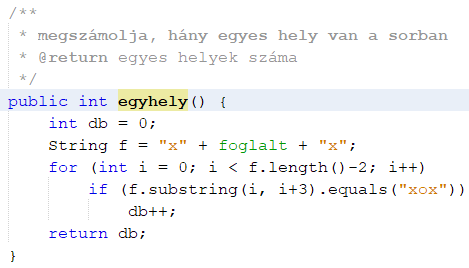
Próbáld ki! Mennyi lenne a teljes bevétel? Írd ide:

## Feladat

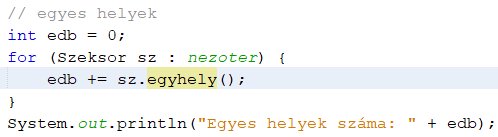
Mivel az emberek általában nem egyedül mennek színházba, ha egy üres hely mellett nincs egy másik üres hely is, akkor azt nehezebben lehet értékesíteni. Határozd meg, és írasd ki a képernyőre, hogy hány ilyen „egyedülálló” üres hely van a nézőtéren!

## Megoldás

Szeksor.java:



Szinhaz.java:



# Források

* Cay Horstmann: Big Java, Early Objects  
  John Wiley & Sons, Inc., 2015  
  ISBN 978-1-119-05628-7
* Informatika és informatikai ismeretek érettségi feladatok