Tömbök és tömbök bejárása

Több, azonos típusú adatot egy közös csoportban, egy *tömb*ben is tudunk kezelni. A java-ban tömbök esetén fontos, hogy azonos típusú adatokról van szó (vannak nyelvek, ahol nem).

Java-ban tömböt tartalmazó változót úgy deklarálunk, hogy a típus neve után szögletes zárójel párt teszünk.

```
jshell> int[] x
x ==> null
```

Az **x** most egy olyan változó, amely egész számokat tartalmazó tömb tud lenni. De amint a második sorban látjuk, az **x** értéke most **`null`**. Ebből az is látszik, hogy a tömb nem egy primitív típus, aminek alapértelmezett értéke lenne (mint például int esetén 0).

Hogy ténylegesen használni is tudjuk ezt a tömböt, inicializálnunk is kell. Inicializáláskor pedig meg kell adnunk a tömb hosszát, vagyis azt, hogy hány elem tárolására lesz majd képes.

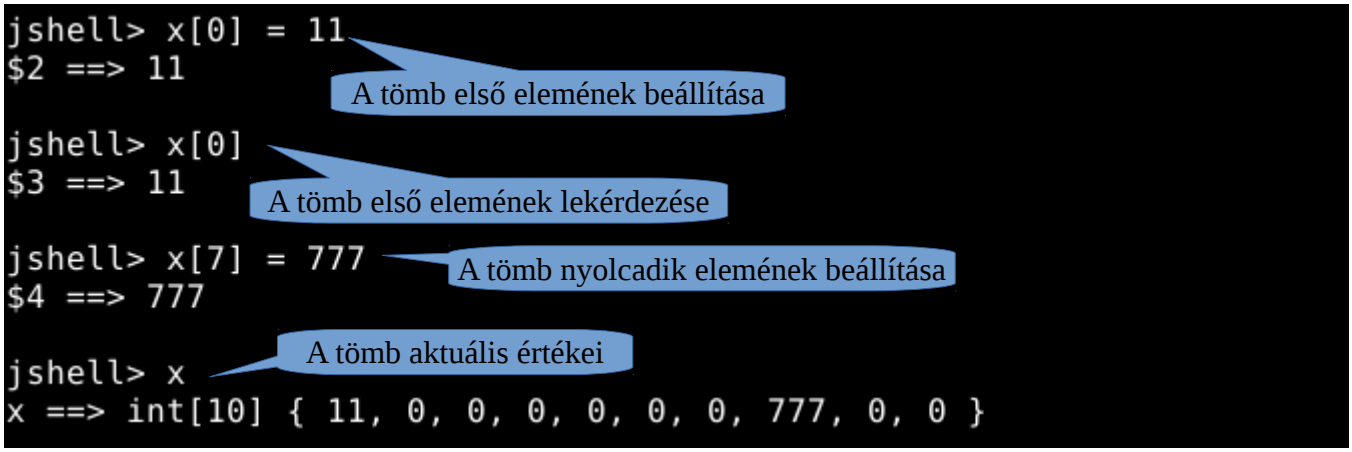
```
jshell> x = new int[10]
x ==> int[10] { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }
```

Az **x** változót úgy inicializáltuk, hogy 10 elemű legyen, vagyis 10 egész szám tárolására legyen képes. Azt is látjuk, hogy ezeket az értékeket a rendszer 0-ra inicializálta (ez az int alapértelmezett értéke).

A tömb **hosszát** inicializálás után nem lehet módosítani (csak újrainicializálással, de ilyenkor a korábbi tömb gyakorlatilag elveszik). Ez azért van, mert inicializáláskor foglal le a rendszer memóriaterületet a tömbnek. (Léteznek összetettebb tömb-szerű adatszerkezetek, amik dinamikusan bővíthetők → *List*.)

A tömb elemeit a sorszámuk segítségével érjük el. A tömbök elemeinek sorszáma 0-val kezdődik (ez az első). A 10 elemű tömb utolsó eleme tehát a 9-es indexet birtokolja. Az elemek sorrendje rögzített, véletlenszerűen nem változik.

A tömb egyes elemei változóként működnek, vagyis lehet őket módosítani a **=** operátorral.



A tömb hosszát le lehet kérdezni a **`.length`** attribútum segítségével. Ez nem egy függvénye a tömbnek, ezért nem kell zárójeleket tenni utána (ha zárójelet teszünk, hibás lesz). A tömb hossza ezzel sem változtatható meg.

A tömb inicializálása és kezdő értékeinek beállítása egy lépésben is történhet a **`{}`** segítségével. Ilyenkor csak felsoroljuk az elemeket, a tömb hosszát a környezet automatikusan beállítja.

```
jshell> int[] y = {1, 2, 3, 0, 0, 0}
y ==> int[6] { 1, 2, 3, 0, 0, 0 }

jshell> y.length
$7 ==> 6
```

Tömbbe tömböket is lehet tenni, de továbbra is igaz, hogy az elemek típusai egyformák kell legyenek (amit deklaráláskor megadtunk). Ezt nevezik egyébként *több dimenziós* tömbnek. Az elnevezés az elemek elérése módjából érthető.

```
jshell> String[][] ketdimenzios_szoveg_tomb = {
   ...>   {"1.sor 1.elem", "1.sor 2.elem", "1.sor, 3.elem"},
   ...>   {"2.sor 1.elem", "2.sor 2.elem"},
   ...>   {},
   ...>   {"4.sor 1.elem", "4.sor 2.elem", "4.sor, 3.elem"}
   ...> }
ketdimenzios_szoveg_tomb ==> String[4][] { String[3] { "1.sor 1.ele
m", "1.sor  ... elem", "4.sor, 3.elem" } }

jshell> ketdimenzios_szoveg_tomb[1][1]
$13 ==> "2.sor 2.elem"

jshell> ketdimenzios_szoveg_tomb[3].length
$14 ==> 3

jshell> ketdimenzios_szoveg_tomb[2].length
$15 ==> 0
```

Amiket a fenti példából megfigyelhetünk:

- A tömbön belüli tömbök hossza nem kell azonos legyen (a fenti példában az első és utolsó al-tömb hossza 3, a másodiké 2, a harmadik pedig üres).
- A tényleges értékeket a **`[i][j]`** indexeléssel érjük el.
- Az elemek számozása itt is 0-val kezdődik, így érthető, hogy az **`[1][1]`** mért a ”2.sor 2.elem”.
- A **`[i]`** indexeléssel magukat az al-tömböket érjük el (ezeknek van **`.length`** tulajdonsága).

Mint korábban láttuk, ha egy tömböt nem inicializálunk (csak deklarálunk), az értéke **`null`** lesz. Van olyan is, hogy üres tömb (**`new int[0]`** vagy **`{}`**), de a kettő nem ugyanaz. Lényeges különbség, hogy míg az üres tömbnek van hossza (ami természetesen 0), a null-nak nincs.

```
jshell> int[][] n = {null, {}}
n ==> int[2][] { null, int[0] {  } }

jshell> n[0].length
|  java.lang.NullPointerException thrown:
|        at (#18:1)

jshell> n[1].length
$19 ==> 0
```

Ez itt hiba!

Ez rendben van

A **`.length`** létezése azért fontos, mert a legtöbb esetben ennek a segítségével dolgozzuk fel, járjuk be a tömböt (elkerülendő a túlindexelés hibáját). Még konkrétabban: amikor egy függvény bemeneti paraméterként egy tömböt vár, aminek a tényleges hossza hívásonként eltérő lehet, nem szokták külön paraméterben tudatni a tömb hosszát (egyrészt azért, mert ez amúgy is lekérdezhető, másrészt azért, mert hiba esetén el is térhet).

Tömb bejárása

Egy tömb bejárásáról, végigjárásáról olyankor beszélünk, amikor egy adott műveletet a tömb minden egyes elemére el akarunk végezni. Akár olyan egyszerű műveletet, mint az adott elem kiírása a kimenetre.

```
jshell> int[] tomb = { 1, 2, 3 }
tomb ==> int[3] { 1, 2, 3 }

jshell> for(int elem : tomb) {
   ...>   System.out.println("elem: " + elem);
   ...> }
elem: 1
elem: 2
elem: 3
```

A fenti bejárás egyes ciklusaiban az **`elem`** változó a **`tomb`** tömb egyes elemeinek értékét veszi fel sorra. Fontos, hogy a bejáró elem típusa azonos kell legyen a tömb elemeinek típusával. A változó neve természetesen tetszőleges.

A fenti példában nem szembetűnő, hogy a ciklus miként használja a tömb hosszát (**`length`**), de a háttérben használja. Ha inicializálatlan tömböt adunk egy ilyen bejárásnak bemenetként, hibát okoz (tessék kipróbálni).

A bejárás „klasszikus” példájában azonban már sokkal nyilvánvalóbb a **`length`** használata:

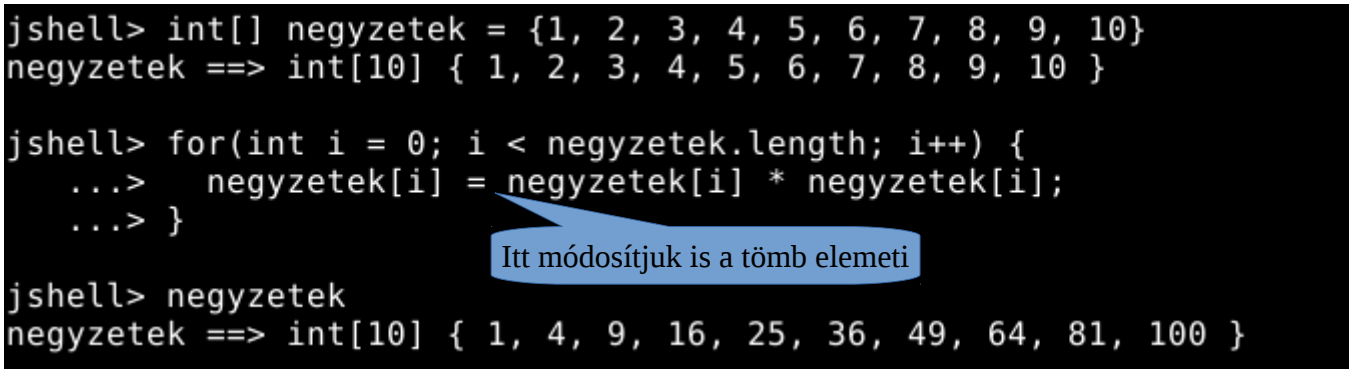
```
jshell> for(int i = 0; i < tomb.length; i++) {
   ...>   System.out.println("elem: " + tomb[i]);
   ...> }
elem: 1
elem: 2
elem: 3
```

Ebben a formában a ciklus szoros értelemben nem is a tömb elemein megy végig, hanem az indexein. A ciklusváltozó itt nem olyan elem, ami azonos típusú a tömb elemeinek típusával és az egyes lépésekben egy tömbbeli elem értékét veszi fel, hanem egy egész típusú változó (a fenti példában **`i`**), amely a ciklus egyes lépéseiben azt mutatja, hogy épp *hányadik* elem vizsgálatánál tartunk. Ebből az is következik, hogy a ciklusmagban a vizsgált elem konkrét értékét az index segítségével érjük el (a fenti példában **`tomb[i]`**).

Bár valamivel összetettebb ez a módszer, több lehetőséget is biztosít. Ezek közül a leglényegesebbek:

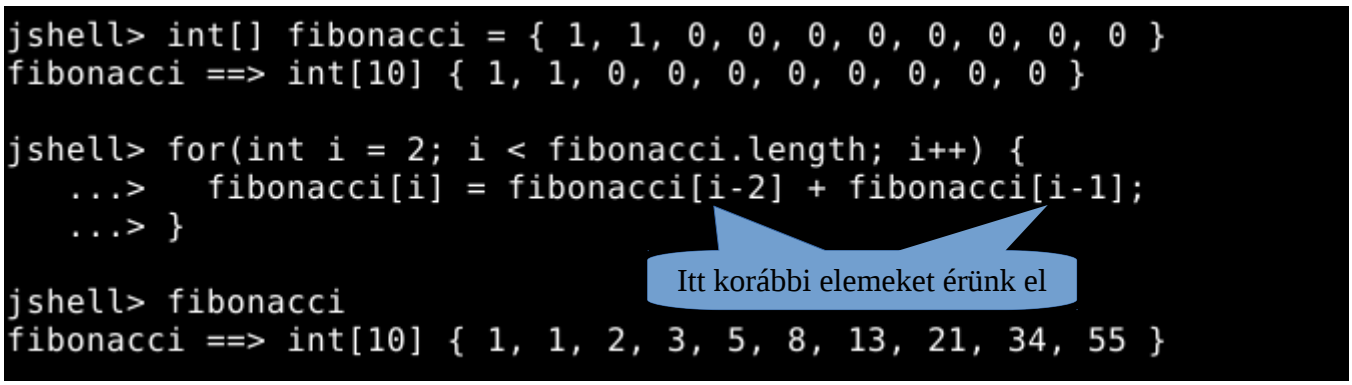
1. Mivel az iteráció egyes lépéseiben nem csak a egyes elemeket kapjuk meg, hanem az index is tudott, lehetőségünk van a tömb módosítására (figyelem, itt nem az elemet módosítjuk, hanem a tömböt!).

Példa:



2. Lehetőségünk van az index alapján a szomszédos elemeket is elérni (a határokon a túlindexelésre persze vigyázni kell).

Példa:



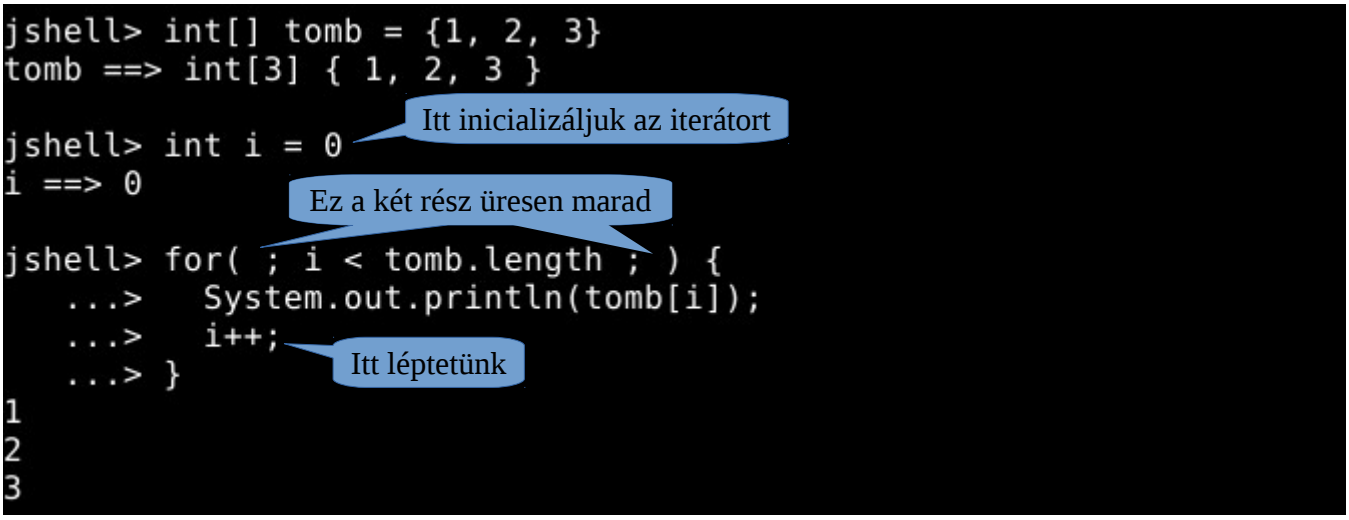
A túlindexelésre való vigyázás a fenti példában azt jelenti, hogy ha kettővel korábbi elemet is el akarok érni, akkor a ciklust nem az első elemnél kezdjük, hanem a harmadiknál. Az első elem előtt kettővel ugyanis „semmi sincs” (magyarul: **`ArrayIndexOutOfBoundsException`**).

3. Nagyobb a szabadságunk abban is, hogy honnan kezdjük a bejárást (a fenti példában a harmadik elemtől kezdtük), hogy meddig folytassuk a bejárást, sőt az sem kötött, hogy minden egyes elemen végigmenjünk. Ehhez vizsgáljuk meg közelebbről a **`for(…) {…}`** paramétereit.

Itt ;-vel elválasztva három részt látunk. Az első rész az inicializálás, rendszerint itt adjuk meg, hogy hányadik indextől induljon a bejárást. A második rész egy feltételvizsgálat, ami azt határozza meg, hogy meddig ismétlődjék a ciklus (tipikus esetben: amíg az index el nem éri a tömb végét; itt jó észben tartani, hogy az utolsó index a tömb hosszánál egyel kisebb). A harmadik rész azt határozza meg, hogy miként léptesse az indexet a ciklusok között (a ++ az egyel növelés operátora).

Az első és a harmadik tag opcionális. Ilyenkor természetesen külön kell gondoskodnunk a ciklusváltozó inicializálásáról, ill. léptetéséről, tágabb értelemben arról, hogy a második részben megadott feltétel előbb-utóbb hamissá váljon.

Erre példa:



A fenti példa egyébként gyakorlatilag egyenértékű a **`while`** ciklussal:

```
jshell> int[] tomb = {1, 2, 3}
tomb ==> int[3] { 1, 2, 3 }

jshell> int i = 0
i ==> 0

jshell> while(i < tomb.length) {
   ...>   System.out.println(tomb[i]);
   ...>   i++;
   ...> }
1
2
3
```

A **`while`** ciklusnál csak a ciklus ismétlésének feltételét kell meghatározni (ami a **`for`** ciklus második tagja szokott lenni), arról külön kell gondoskodni, hogy ez a feltétel előbb-utóbb hamissá váljon, s érjen véget a ciklus. Különben lesz egy vagány végtelen ciklusunk.

A **`while`** ciklust egyébként tipikusabban olyan eseteknél használjuk, amikor előre nem látható, hogy meddig kell ismételni a feldolgozást (egy tömb bejárásánál ez azért előre sejthető). Jellemző eset például amikor egy szövegfájlt olvasunk be soronként (a beolvasást meg addig kell ismételni, amíg még van mit olvasni), egy fájl megnyitásánál ugyanis nem triviális, hogy a hány sort tartalmaz.

Feladatok

1. Készíts egy három elemű tömböt, aminek az elemei szöveg típusúak. Járd végig és írd ki az értékeit a kimenetre.

2. Készíts egy öt elemű tömböt, aminek az elemei valós típusúak. Járd végig a tömböt és az értékeit add össze. A kimenetre az elemek összegét írd ki.

3. Készíts egy tíz elemű tömböt egész számokkal. Járd végig a tömböt és írd ki a kimenetre a páros számú elemek értékét.

4. Készíts egy tíz elemű tömböt egész számokkal, és írd ki a képernyőre az első páros számot (ha ilyent találsz, ne járd végig tovább a tömböt).

5. Készíts egy 3x3-as egészekből álló kétdimenziós tömböt a következő értékekkel:

1 2 3
4 5 6
7 8 9

Készíts egy ciklust, ami bal felsőtől a jobb alsóig menő átlóban levő értékeket írja ki (1, 5, 9).

6. A szöveg típusú objektumok (String) **`split`** függvénye egy reguláris kifejezés alapján feldarabolja a szöveget, visszaadva egy tömböt. Darabolj fel egy szöveget a szavaira (szóközök alapján), majd írd ki ezeket egyenként, külön sorba.

7. Az előbbihez hasonlóan darabolj fel egy szöveget szavakra, de csak a neveket írd ki (ami nagybetűvel kezdődik).

8. Írd ki egy tömb elemeit visszafelé.

9. Írd ki egy tömb minden második elemét.

10. Döntsd el egy egészekből álló tömbről, hogy a páros vagy a páratlan indexű elemeinek az összege a nagyobb (vagy egyenlő).