A felszín alatt

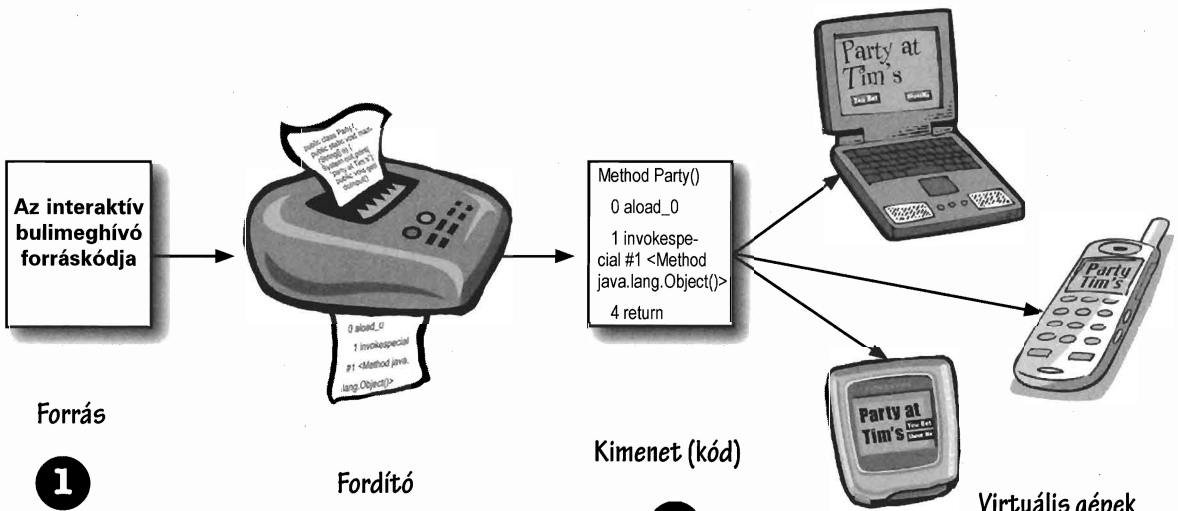


A Java új helyekre visz. A szerény kezdetek óta – amikor a nyilvánosság megismerte az (eléggé zöldfülű) 1.02-es változatot – a Java sok-sok programozót csábított el barátságos nyelvtanával, objektumközpontú szolgáltatásaival, memóriakezelésével, és mind közül a legvonzóbb vonásával – a hordozhatóság ígéretével. A **„megírod egyszer, futtathatod mindenhol"** csábításának egyszerűen nem lehet ellenállni. A lelkes követők hada egyre nőtt, bár a programozóknak meg kellett küzdeniük a hibákkal, a korlátokkal, és igen, azzal a ténnyel, hogy a Java lassú, mint egy másnapos csiga. Persze ez már a múlté. Ha csak most ismerkedsz a Javával, **szerencsés vagy**. Vannak köztünk olyanok, akiknek mérföldeket kellett gyalogolniuk a hóban, hegyre fel és le (mezítláb), hogy akárcsak a legegyszerűbb kisalkalmazást működésre bírják. *Neked* viszont a mai, **karcsúbb, gyorsabb és sokkal nagyobb tudású** Java jutott.



Így működik a Java

A cél az, hogy írj egy alkalmazást (ebben a példában egy interaktív bulimeghívót), és működésre bírd azokon az eszközökön, amikkel a barátaid rendelkeznek.



Forrás

1

Írd meg a forrásdokumentumot egy ismert nyelv (ebben az esetben a Java) segítségével.

Fordító

2

Futtasd át a dokumentumot egy forráskódfordítón. A fordító (compiler) ellenőrzi, hogy nincsenek-e hibák, és addig nem végzi el a fordítást, amíg meg nem győződött róla, hogy minden helyesen fog lefutni.

Kimenet (kód)

3

A fordító létrehoz egy Java ***bájtkódot*** tartalmazó új dokumentumot. Ezt a fájlt minden Java-kód futtatására képes eszköz értelmezni tudja, illetve át tudja alakítani valamilyen általa futtatható alakra.
A lefordított bájtkód rendszerfüggetlen.

Virtuális gépek

4

A barátaidnak nincs fizikai, megfogható Java-gépe, de mindnyájan rendelkeznek egy (szoftverrel megvalósított) ***virtuális*** Java-géppel, ami az elektronikus kütyüjükön fut.
A virtuális gép elolvassa és *lefuttatja* a bájtkódot.

A te dolgod a Javában

Begépeled a kódot a forrásfájlba, lefordítod a javac fordítóval, majd a lefordított bájtkódot lefuttatod egy Java virtuális gépen.

```
import java.awt.*;
import java.awt.event.*;
class Party {
  public void buildInvite() {
    Frame f = new Frame();
    Label l = new Label("Party at Tim's");
    Button b = new Button("You bet");
    Button c = new Button("Shoot me");
    Panel p = new Panel();
    p.add(l);
  } // more code here...
}
```

Forrás

1

Gépeld be a forráskódot.

Mentsd ***Party.java*** néven.

```
File Edit Window Help Plead
%javac Party.java
```

Fordító

2

Fordítsd le a Party.java fájlt a `javac` (a fordító-program) futtatásával. Ha nem merül fel hiba, egy ***Party.class*** nevű második dokumentumot kapsz eredményül.

A fordító által előállított Party.class fájl *bájtkódot* tartalmaz.

```
Method Party()
  0 aload_0
  1 invokespecial #1 <Method
java.lang.Object()>
  4 return
Method void buildInvite()
  0 new #2 <Class java.awt.Frame>
  3 dup
  4 invokespecial #3 <Method
java.awt.Frame()>
```

Kimenet (kód)

3

A lefordított kód: ***Party.class***.

```
File Edit Window Help Swear
%java Party
```



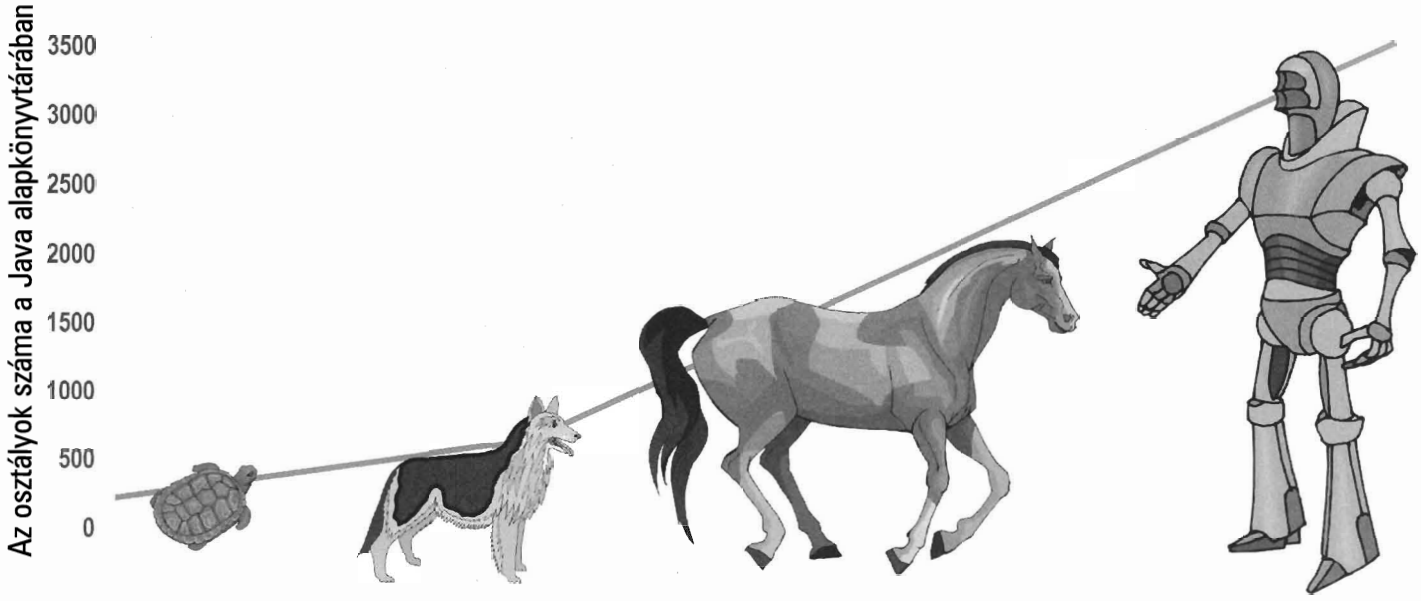
Virtuális gépek

4

Futtasd a programot: indítsd el a Java virtuális gépet (Java Virtual Machine, JVM) a ***Party.class*** fájllal. A JVM a *bájtkódot* olyan alakra hozza, amit a háttérben működő rendszer is megért, és lefuttatja a programod.

Megjegyzés: ezt nem oktatóanyagnak szántuk... Mindjárt igazi kódot fogsz írni, csak azt akartuk, hogy először ráérezz, hogyan illenek össze a kirakós darabjai.

A Java nagyon rövid története



Java 1.02

250 osztály
Lassú.
Aranyos név és embléma.
Szórakoztató a használata. Sok hiba.
A Nagy Durranást a kisalkalmazások (***applet***-ek) jelentik.

Java 1.1

500 osztály
Kicsit gyorsabb.
Nagyobb tudású, ***barátságosabb***.
Kezd nagyon ***népszerűvé*** válni.
Jobb GUI-kód.

Java 2 (1.2–1.4 változatok)

2300 osztály
Sokkal gyorsabb.
(Néha) akár a hagyományos programok sebességét is eléri.
Komoly, ***sokoldalú***. Háromféle ízben kapható: Micro Edition (J2ME), Standard Edition (J2SE) és Enterprise Edition (J2EE). Az új (különösen a webes alapú) üzleti és mobilalkalmazások ***kedvenc nyelve***.

Java 5.0 (1.5 és újabb változatok)

3500 osztály
Még nagyobb tudás; könnyebb vele fejleszteni.
Az ezernél is több új osztály bevezetésén túl a Java 5.0 (Tiger, vagyis a „Tigris") magát a nyelvet is jelentősen megváltoztatta: a programozók számára (legalábbis elméletben) könnyebben használhatóvá vált, és olyan új szolgáltatásokat kínál, amelyek más nyelvekben már nagy népszerűségre tettek szert.

Nézd, milyen könnyű Java-kódot írni!

```
int size = 27;
String name = "Fido";
Dog myDog = new Dog(name, size);
x = size - 5;
if (x < 15) myDog.bark(8);

while (x > 3) {
   myDog.play();
}

int[] numList = {2,4,6,8};
System.out.print("Hello");
System.out.print("Dog: " + name);
String num = "8";
int z = Integer.parseInt(num);

try {
   readTheFile("myFile.txt");
}
catch(FileNotFoundException ex) {
   System.out.print("File not found.");
}
```

Hegyezd ki a ceruzád!

Próbáld kitalálni, hogy mit csinálnak az egyes kódsorok! (A válaszok a következő oldalon találhatók.)

bevezet egy egész típusú, „size" nevű változót, és hozzárendeli a 27 értéket

K: Látom, hogy van Java 2 és Java 5.0, de volt Java 3 és 4 is? És miért Java 5.*0*, ha Java 2, *0* nélkül?

V: A piaci szakemberek furfangja… Amikor a Java 1.1-ről 1.2-re váltott, olyan drámai változások történtek a nyelvben, hogy a marketingesek úgy érezték, teljesen új „névre" van szükség, ezért az újszülöttet ***Java 2***-nek kezdték hívni, bár a tényleges változatszám szerint Java 1.2-ről volt szó. Az 1.3-as és 1.4-es változatokat szintén a ***Java 2*** kiadásainak tekintették, Java 3 vagy 4 azonban *soha* nem létezett. A Java 1.5-ös változatánál a piackutatók ismét úgy döntöttek, hogy a változások olyan horderejűek, hogy új név szükségeltetik (ezzel a legtöbb fejlesztő is egyetértett), ezért számbavették a lehetőségeket. A sorban a következő szám a „3" lett volna, de ha a Java 1.5-nek a ***Java 3*** nevet adják, az még jobban összezavarta volna az embereket, ezért a ***Java 5.0*** név mellett tették le a voksukat, hogy legalább valami megmaradjon az 1.5-ös változatszám 5-öséből.
Az eredeti Java (az első hivatalos kiadás) tehát az 1.02-es változat volt, és az 1.1-es változatig egyszerűen „Javának" hívták a nyelvet. Az 1.2-es, 1.3-as és 1.4-es változatok a „Java 2" nevet viselték, az 1.5-ös változattól kezdve pedig „Java 5.0"-ról beszélünk, de találkozhatunk a („.0" nélküli) Java 5 és a „Tiger" névvel is (ez volt a Java 5.0 eredeti kódneve) – ezek ugyancsak a Java 5.0-t jelölik. Azt megtippelni sem merjük, hogy milyen nevet adnak a *következő* kiadásnak…



Hegyezd ki a ceruzád! Megoldás

Nézd, milyen könnyű Java-kódot írni!

Ne aggódj, ha az alábbiakból még semmit sem értesz!
Mindent, ami itt szerepel, részletesen elmagyarázunk a könyvben – a többségüket még az első 40 oldalon. Ha a Java hasonlít valamelyik általad ismert nyelvre, akkor bizonyos dolgokat könnyűnek fogsz találni. Ha nem, akkor sem kell aggódnod. *Előbb-utóbb sort kerítünk rá...*

`int size = 27;`	bevezet egy egész típusú, „size" nevű változót, és hozzárendeli a 27 értéket
`String name = "Fido";`	bevezet egy karakterlánc típusú, „name" nevű változót, és hozzárendeli a „Fido" értéket
`Dog myDog = new Dog(name, size);`	bevezet egy új, Dog (Kutya) típusú, „myDog" nevű változót, az új kutyához hozzárendeli a „name" és „size" változókat
`x = size - 5;`	kivon 5-öt a 27-ből (a „size" értékéből), és hozzárendeli az eredményt egy „x" nevű változóhoz
`if (x < 15) myDog.bark(8);`	ha az x (aminek az értéke 22) kisebb 15-nél, a kutyát arra utasítja, hogy ugasson (bark) 8-szor
`while (x > 3) {`	a ciklus addig ismétlődik, amíg x nagyobb 3-nál...
`  myDog.play();`	arra utasítja a kutyát, hogy játsszon (play) – akármit jelentsen is ez egy kutyánál...
`}`	ez úgy néz ki, mintha a ciklus vége lenne – minden, ami a kapcsos zárójelek között áll, a ciklus része
`int[] numList = {2,4,6,8};`	bevezet egy egészek listájából álló, „numList" nevű változót, és beleteszi a 2,4,6,8 értékeket a listába
`System.out.print("Hello");`	kiírja a „Hello" szöveget...valószínűleg a parancsablakban
`System.out.print("Dog: " + name);`	kiírja a „Hello Fido" szöveget (a „name" értéke „Fido") a parancsablakban
`String num = "8";`	bevezet egy karakterlánc típusú, „num" nevű változót, és hozzárendeli a „8" értéket
`int z = Integer.parseInt(num);`	a „8" karakterláncot a 8 valódi számértékké alakítja
`try {`	„megpróbálunk" (try) valamit...szóval nem biztos, hogy működik...
`  readTheFile("myFile.txt");`	beolvasunk (legalábbis MEGPRÓBÁLUNK beolvasni) egy „myFile.txt" nevű szövegfájlt
`}`	itt lehet a vége a „próbálkozásnak", vagyis valószínűleg több dologgal is próbálkozhatunk...
`catch(FileNotFoundException ex) {`	biztos itt derül ki, hogy működik-e, amit csinálni próbáltunk...
`  System.out.print("File not found.");`	ha a kipróbált művelet nem sikerült, a „File not found" (A fájl nem található) szöveg jelenik meg a parancsablakban
`}`	úgy fest, a kapcsos zárójelek közötti rész adja meg, hogy mit kell csinálni, ha a „próbálkozás" nem jött össze...

A Java-kódok felépítése



Elhelyezünk egy osztályt egy forrásfájlban.

Az osztályba tagfüggvényeket teszünk.

A tagfüggvényekben utasításokat helyezünk el.

Mi kerül a forrásfájlokba?

A (`.java` kiterjesztésű) forráskódfájlok egy-egy ***osztály*** (class) meghatározását tárolják. Az osztály a programod egy *darabját* jelképezi, bár egy aprócska alkalmazás állhat egyetlen osztályból is. Az osztályt kapcsos zárójelek között kell meghatároznod.

```
public class Dog {



}
```

osztály

Mi kerül az osztályokba?

Az osztályok egy vagy több ***tagfüggvényből*** (method) állnak. A Dog osztályban a ***bark*** tagfüggvény arra nézve tartalmaz utasításokat, hogy egy kutyának hogyan kell ugatnia. A tagfüggvényeidet az osztályokon *belül* (vagyis az osztály kapcsos zárójelei között) kell bevezetned.

```
public class Dog {
   void bark() {



   }
}
```

tagfüggvény

Mi kerül a tagfüggvényekbe?

A tagfüggvények kapcsos zárójelei között írhatod le azokat az utasításokat, amelyek megadják a tagfüggvény végrehajtásának módját. A tagfüggvények *kódja* lényegében utasítások sorából áll; egyelőre gondolhatsz úgy a tagfüggvényekre, mintha normál függvények vagy eljárások lennének.

```
public class Dog {
   void bark() {
      utasítás1;
      utasítás2;
   }
}
```

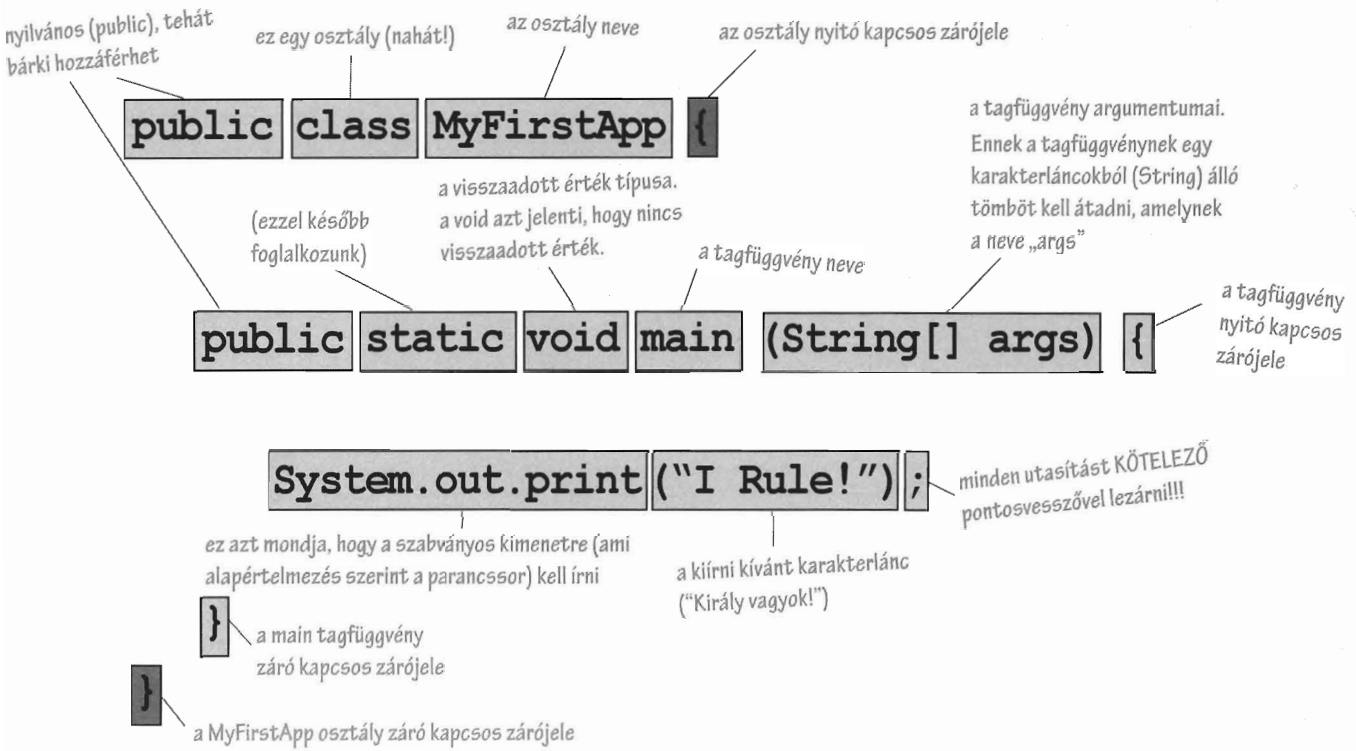
utasítások

Az osztályok felépítése

Amikor a JVM elindul, felkutatja azt az osztályt, amelyiknek a nevét megadtad neki a parancssorban, majd egy különleges alakú tagfüggvényt keres, ami pontosan így néz ki:

```
public static void main (String[] args) {
   // ide jön a kódod
}
```

Ezt követően a JVM mindent lefuttat, ami a `main` („fő") tagfüggvény kapcsos zárójelei között található. Minden Java-alkalmazásban lennie kell legalább egy **osztálynak**, és legalább egy **`main`** tagfüggvénynek (nem *osztályonként*, csak *alkalmazásonként*).



Most még nem kell megjegyezned semmit...
ez a fejezet csak a kezdő lépések megtételében segít.

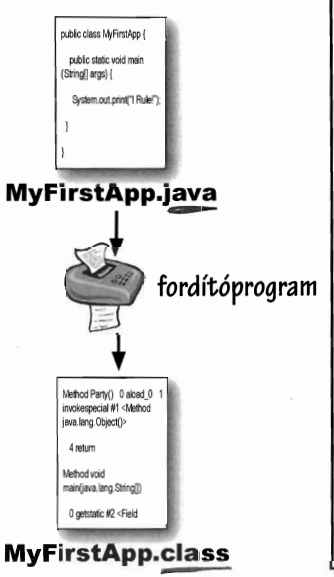
Osztály, main tagfüggvénnyel

A Javában mindent osztályokba kell tenned. Begépeled a kódot egy (*.java* kiterjesztésű) forrásfájlba, majd a fordítóval előállíttatsz belőle egy új (*.class* kiterjesztésű) osztályfájlt. Amikor lefuttatod a programot, valójában egy *osztályt* futtatsz.

Egy program futtatása azt jelenti, hogy utasítod a JVM-et (Java Virtual Machine), hogy „töltse be a **`MyFirstApp`** osztályt, majd hajtsa végre annak **`main()`** tagfüggvényét, és futtassa addig, amíg az abban található összes kóddal nem végzett".

A 2. fejezetben mélyebben is elmerülünk az osztályok világában, egyelőre azonban elég azon gondolkodnod, hogy ***miként írhatsz futtatható Java-kódot***. Ehhez pedig az első lépést mindig a **main()** tagfüggvény jelenti.

Egy program mindig a **main()** tagfüggvénnyel kezdi meg a futását. Nem számít, hogy milyen nagy a programod (vagyis hogy hány *osztályt* használ), az elinduláshoz muszáj lennie egy **main()** tagfüggvénynek.




```
public class MyFirstApp {

   public static void main (String[] args) {
     System.out.println("I Rule!");
     System.out.println("The World");
   }

}
```

1 Mentés

```
MyFirstApp.java
```

2 Fordítás

```
javac MyFirstApp.java
```

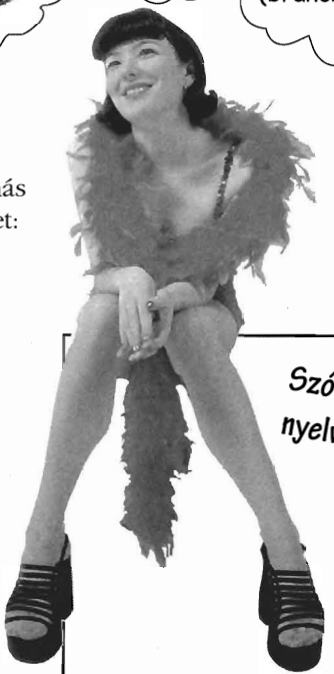
3 Futtatás

```
File Edit Window Help Scream
%java MyFirstApp
I Rule!
The World
```

Mit mondhatsz el a main tagfüggvényben?



A játék akkor kezdődik, amikor a main (vagy *bármely másik*) tagfüggvényen belülre kerülsz. Ahhoz, hogy *a számítógép csináljon valamit*, ugyanazokat a szokásos dolgokat mondhatod el ott, mint más programozási nyelvekben. A kódod az alábbiakra utasíthatja a JVM-et:



1 csinálj valamit

Utasítások: bevezetések (deklarációk), értékadások, tagfüggvényhívások stb.

```
int x = 3;
String name = "Dirk";
x = x * 17;
System.out.print("x is " + x);
double d = Math.random();
//    ez egy megjegyzés
```

2 csinálj valamit újra és újra

Ciklusok: *for* és *while*

```
while (x > 12) {
   x = x -1;
}

for (int x = 0; x < 10; x = x + 1) {
   System.out.print("x is now " + x);
}
```

3 csinálj valamit, ha ez a feltétel teljesül

Elágaztatás: *if/else* vizsgálatok

```
if (x == 10) {
   System.out.print("x must be 10");
} else {
   System.out.print("x isn't 10");
}
if ((x < 3) & (name.equals("Dirk"))) {
   System.out.println("Gently");
}
System.out.print("this line runs no matter what");
```

Szórakoztató nyelvtan

- Minden utasítás végén pontosvesszőnek kell állnia.
  ```
  x = x + 1;
  ```
- Az egysoros megjegyzések két perjellel kezdődnek.
  ```
  x = 22;
  // ez a sor idegesít
  ```
- A térközök általában nem számítanak.
  ```
  x   =    3  ;
  ```
- A változókat (variable) egy **névvel** és egy **típussal** kell bevezetni (a 3. fejezetben az összes *Java-típussal* megismerkedsz majd).
  ```
  int weight;
  // típus: int, név: weight
  ```
- Az osztályokat és tagfüggvényeket kapcsos zárójelek között kell meghatározni.
  ```
  public void go() {
  // ide jön a csodálatos
      ➥ kódod
  }
  ```

```
while (moreBalls == true) {
  keepJuggling();
}
```

folytatZsonglorkodes();



Ismétlődő ciklusok ismétlődő ciklusok ismétlődő...

A Javában három szabványos ciklusszerkezet létezik: a *while*, a *do-while* és a *for*. A ciklusokról mindent megtudhatsz a könyv későbbi részében, de az még odébb van, úgyhogy egyelőre maradjunk a *while*-nál. Az utasításforma (nem beszélve a logikáról) annyira egyszerű, hogy valószínűleg már be is aludtál rajta: amíg („while") valamilyen feltétel igaz, mindent újra és újra végre kell hajtani, ami a ciklus *törzsében* áll. A ciklustörzs határait kapcsos zárójelek jelzik, tehát mindent, amit ismételgetni szeretnél, a köztük álló tömbben (blokkban) kell elhelyezned. A ciklusok kulcsa a *feltételvizsgálat*. A Javában a feltételvizsgálat egy olyan kifejezés, ami *logikai értéket* (Boole-féle értéket) eredményez – vagyis valami olyasmit, ami vagy igaz (***true***), vagy hamis (***false***). Ha valami olyasmit mondasz, hogy „While *fagyiATolcserben true*, nyalj tovább", akkor világos, hogy egy ilyen kétértékű logikai vizsgálatot kell elvégezni: vagy *van* fagyi a tölcsérben, vagy *nincs*. Ha viszont azt mondod, hogy „While *Bob*, nyalj tovább", akkor nem valódi vizsgálatról van szó. Ahhoz, hogy ez működjön, valami ilyesmire kell változtatnod: „While Bob durmol..." vagy „While Bob *nem* kockásat visel...".

Egyszerű logikai vizsgálatok

Egyszerű logikai vizsgálatot végezhetsz például úgy, hogy a következő *összehasonlító műveletekkel* ellenőrzöd egy változó értékét:

< (kisebb, mint)
> (nagyobb, mint)
== (egyenlő) (igen, ez *két* egyenlőségjel)

Figyeld meg a különbséget az *értékadó* művelet (*egyetlen* egyenlőségjel) és az *egyenlő* művelet (*két* egyenlőségjel) között! Sok programozóval előfordul (persze veled nem), hogy véletlenül azt írja, hogy **=**, amikor valójában *azt akarja*, hogy **==**.

```
int x = 4; // a 4 érték hozzárendelése az x-hez
while (x > 3) {
  // a ciklus kódja lefut, mert
  // x nagyobb, mint 3
  x = x - 1; // vagy örökké ismétlünk
}
int z = 27; //
while (z == 17) {
  // a ciklus kódja nem fut le, mert
  // z nem egyenlő 17-tel
}
```

Nincsenek hülye kérdések

K: Miért kell mindent osztályba tenni?

V: A Java objektumközpontú (objektumorientált, OO) nyelv. Nem úgy működik, mint régen, amikor gőzmeghajtású fordítókkal kellett dolgoznunk, és egyetlen tömbszerű forrásfájlt írtunk egy halom eljárással. A 2. fejezetben megtanulod majd, hogy minden osztály egy objektum sablonja, és a Javában szinte minden objektum.

K: Minden osztályomban el kell helyeznem egy main tagfüggvényt?

V: Dehogy. Egy Java-program osztályok tucatjait használhatja, miközben esetleg csak az *egyikben* van main tagfüggvény – az, amelyik elindítja a program futását. Ugyanakkor írhatsz például tesztosztályokat, amelyeknek a main tagfüggvénye *más* osztályokat ellenőriz.

K: A másik programozási nyelven, amit ismerek, egész értékeken is lehet logikai vizsgálatot végezni. A Javában lehet olyasmit mondani, mint az alábbi?

```
int x = 1;
while (x) { }
```

V: Nem. A *boolean* és az *integer* a Javában nem összeegyeztethető típusok. Mivel egy feltételvizsgálat eredményének boolean (logikai) típusúnak *kell* lennie, az egyetlen változótípus, amit közvetlenül (összehasonlító művelet nélkül) vizsgálhatsz, a ***boolean***. Ezt például mondhatod:

```
boolean isHot = true;
while(isHot) { }
```

Példa a `while` ciklusra

```
public class Loopy {
   public static void main (String[] args) {
     int x = 1;                                  "A Ciklus előtt"
     System.out.println("Before the Loop");
     while (x < 4) {                                    "A ciklusban"
       System.out.println("In the loop");
       System.out.println("Value of x is " + x);
         x = x + 1;                          "Az x értéke: " + x
     }
     System.out.println("This is after the loop");
   }                          "Ez már a ciklus után van"
}
```

ez a kimenet

```
% java Loopy
Before the Loop
In the loop
Value of x is 1
In the loop
Value of x is 2
In the loop
Value of x is 3
This is after the loop
```

```
% java Loopy
A Ciklus előtt
A ciklusban
Az x értéke: 1
A ciklusban
Az x értéke: 2
A ciklusban
Az x értéke: 3
Ez már a ciklus után van
```

Újratöltve

- Az utasításokat pontosvesszővel kell lezárnod. **;**
- A kódblokkokat kapcsos zárójelek határolják. **{ }**
- Egy *int* típusú változót a nevével és a típusával vezethetsz be. **int x;**
- Az **értékadó** művelet jele *egyetlen* egyenlőségjel. **=**
- Az **egyenlőség** művelet jele *két* egyenlőségjel. **==**
- Egy *while* ciklus addig futtat mindent, ami a (kapcsos zárójelek által határolt) blokkján belül található, amíg a *feltételvizsgálat* eredménye ***true***.
- Ha a feltételvizsgálat eredménye ***false***, a *while* ciklus kódblokkja nem fut le, a végrehajtás pedig továbbkerül a közvetlenül a ciklusblokk *után* következő kódrészhez.
- A logikai feltételvizsgálatokat kerek zárójelek között adhatod meg:
  `while (x == 4) { }`

Feltételes elágazások

A Javában egy *if* teszt lényegében ugyanaz, mint egy logikai teszt egy *while* ciklusban – azzal a különbséggel, hogy nem azt mondod, hogy „**amíg** (while) van sör…”, hanem azt, hogy „**ha** (if) még van sör…”.

```
class IfTest {
  public static void main (String[] args) {
    int x = 3;
    if (x == 3) {
      System.out.println("x must be 3");
    }
    System.out.println("This runs no matter what");
  }
```

"Az x-nek 3-nak kell lennie"

"Ez mindenképpen lefut"

```
% java IfTest
Az x-nek 3-nak kell lennie
Ez mindenképpen lefut
```

kódkimenet

A fenti kód csak akkor hajtja végre „Az x-nek 3-nak kell lennie” szöveget kiíró sort, ha a feltétel (x egyenlő 3-mal) igaz. Az „Ez mindenképpen lefut” szöveget kiíró sor azonban a feltétel értékétől függetlenül végrehajtódik, vagyis az *x* értékétől függően vagy egy mondat jelenik meg, vagy kettő.

A feltételt azonban kiegészítheted egy *else* résszel, hogy valami ilyesmit mondhass: „*Ha* (if) még van sör, kódolj tovább, *máskülönben* (else) viszont szerezz még sört, és aztán folytasd…”.

```
class IfTest2 {
  public static void  main (String[] args) {
    int x = 2;
    if (x == 3) {
      System.out.println("x must be 3");
    } else {
       System.out.println("x is NOT 3");
    }
    System.out.println("This runs no matter what");
  }
}
```

"Az x-nek 3-nak kell lennie"

"Az x NEM 3"

"Ez mindenképpen lefut"

```
% java IfTest2
Az x NEM 3
Ez mindenképpen lefut
```

új kimenet

System.out.print vagy System.out.println?

Ha figyeltél (és persze hogy figyeltél), akkor felfigyelhettél rá, hogy a **print** helyett a **println** utasításra váltottunk.

Észrevetted a különbséget?

A System.out.***println*** egy új sort szúr be (a print***ln*** a **print*newline***, vagyis „írj új sorba” rövidítése), míg a System.out.-***print*** *ugyanabba* a sorba ír. Ha azt szeretnéd, hogy minden, amit kiíratsz, önálló sorba kerüljön, használd a print**ln** utasítást; ha viszont mindent egy sorban akarsz összeragasztani, maradj a print-nél.

Hegyezd ki a ceruzád!

Adott a következő kimenet:

```
% java DooBee
DooBeeDooBeeDo
```

Pótold a kód hiányzó részeit:

```
public class DooBee {
  public static void main (String[] args) {
    int x = 1;
    while (x < _____ ) {
      System.out._____("Doo");
      System.out._____("Bee");
      x = x + 1;
    }
    if (x == _____ ) {
      System.out.print("Do");
    }
  }
}
```

Komoly üzleti alkalmazás kódolása



Itt az ideje, hogy eddig megszerzett Java-tudásodat a gyakorlatban is kipróbáld. Egy *main()* tagfüggvénnyel rendelkező osztályra lesz szükséged, egy *int* és egy *String* típusú változóra, egy *while* ciklusra, valamint egy *if* tesztre. Egy kis finomítás, és máris felépítheted a címben említett üzleti háttéralkalmazást. *Mielőtt* azonban megnéznéd az ezen az oldalon található kódot, gondolkodj el rajta, *te* hogyan kódolnád a klasszikus amerikai gyerekdalt, a „99 bottles of beer"-t.

```
public class BeerSong {
   public static void main (String[] args) {
     int beerNum = 99;
     String word = "bottles";

     while (beerNum > 0) {

        if (beerNum == 1) {
          word = "bottle"; // egyes szám, mert EGY üveg (bottle)
        }

        System.out.println(beerNum + " " + word + " of beer on the wall");
        System.out.println(beerNum + " " + word + " of beer.");
        System.out.println("Take one down.");
        System.out.println("Pass it around.");
        beerNum = beerNum - 1;

        if (beerNum > 0) {
           System.out.println(beerNum + " " + word + " of beer on the wall");
        } else {
           System.out.println("No more bottles of beer on the wall");
        } // az else vége
     } // a while ciklus vége
  } // a main tagfüggvény vége
} // az osztály vége
```

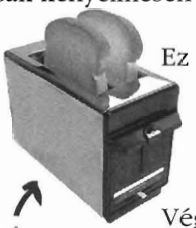
A fenti kódban még van egy kis hiba. Lefordítható és futtatható, de a kimenet nem lesz száz százalékig tökéletes. Ha ki tudod szúrni a hibát, javítsd ki!

Hétfő reggel Bobnál

Bob ébresztőórája hétfőn – mint minden hétköznap – reggel 8:30-kor csörög. Bobnak azonban elég vad hétvégéje volt, ezért a SNOOZE gombra csap – ez pedig életre kelti a Java-képes készülékeket.



Először a vekker küld egy üzenetet a kávéfőzőnek*: „Hé, a gyík megint bealudt, várj a kávéval még 12 percet!" A kávéfőző erre küld egy értesítést a Motorola™ kenyérpirítónak: „Csak kényelmesen a pirítóssal, Bob még durmol!"



Ez után az ébresztőóra Bob Nokia Navigator™ mobiljára küld egy üzenetet: „Hívd fel Bob 9 órás ügyfelét, és mondd meg neki, hogy kicsit késni fog."



Végül a vekker megcsipogtatja Sam (Sam a kutya) drót nélküli nyakörvét a (sajnos már ismerős) jellel, aminek az a jelentése, hogy „Hozd be az újságot, de sétára ne számíts".



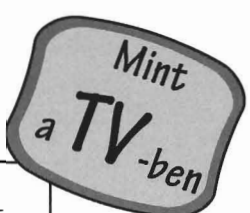
Néhány perccel később a vekker megint megcsörren. Bob ismét SNOOZE-t nyom, a készülékek pedig csevegni kezdenek. Végül az ébresztőóra harmadszor is bepróbálkozik, és amikor Bob a SNOOZE gomb felé nyúl, elküldi Sam nyakörvére az „ugráld körbe és ugass" jelet. Ez végre kiveri az álmot Bob szeméből, úgyhogy kikászálódik az ágyból, hálásan gondolva Java-képességeire és a Radio Shack™-ből összegyűjtött kütyükre, amelyek élménnyé teszik a hétköznap reggeleit.



A pirítós pirul.
A kávé már csorog lefelé.
Az újság ott van a papucsa mellett.

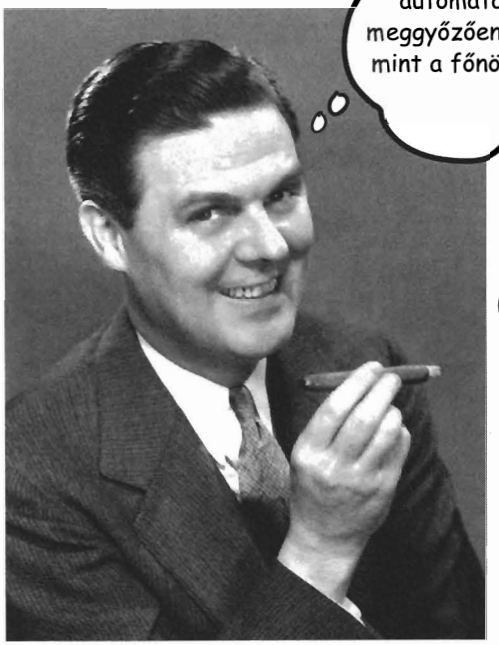
Egy újabb csodálatos reggel ***A Java-képes Házban***!

Neked is lehet Javával felszerelt házad. Nem kell hozzá más, csak egy kis Java, Ethernet és Jini, meg egy személyre szabott megoldás. De óvakodj az úgynevezett „plug and play" („dugd be és játssz" – ami valójában azt jelenti, hogy bedugod, és játszhatsz vele három napig, hogy működésre bírd) vagy „portable" („hordozható") rendszerekre támaszkodó utánzatoktól! Bob nővére, Betty, kipróbált egy ilyen *másik* megoldást, és az eredmény nem volt se meggyőző, se biztonságos. Kár a szegény ebért…



Lehetne ez igaz történet? Igen is, meg nem is. Bár *léteznek* olyan készülékekben működő Java-változatok, mint a PDA-k, a mobiltelefonok (*különösen* a mobiltelefonok), a csipogók vagy az intelligens kártyák, Java-kenyérpirítót vagy -kutyanyakörvet nem biztos, hogy találsz. De még ha nem is bukkansz Java-képes változatra a kedvenc kütyüdből, akkor is működtetheted úgy, *mintha* Java-eszköz lenne, ha valamilyen más felületen (mondjuk a laptopodon) keresztül vezérled, amin *tényleg fut* a Java. Ez az úgynevezett Jini *helyettes architektúra* (Jini surrogate architecture). Igen, *a tiéd is lehet* a kockák álomotthona!

* IP-üzenetszórással (IP multicast), ha kínosan ügyelsz a protokoll-meghatározásra.



Elismerjük, a sörös dalocska nem volt *igazán* komoly üzleti alkalmazás. Még mindig kell valami gyakorlatban alkalmazható dolgot mutatnod a főnöknek? Nézd meg a Frázisautomata kódját!

Megjegyzés: Amikor beírod ezt egy szerkesztőbe, hagyd, hogy a kód saját maga tördelje a szavakat/sorokat! Ne nyomd le az Enter billentyűt, amikor begépelsz egy karakterláncot (olyasmit, ami "idézőjelek" között áll), különben nem lesz lefordítható. Az oldalon látható kötőjelek tehát igaziak, és beírhatod őket, de ne nyomd le az Entert, AMÍG le nem zártad a karakterláncot.

```
public class PhraseOMatic {
   public static void main (String[] args) {
```

1 **// Hozz létre három halmazt, amiből szavakat, illetve kifejezéseket választhatsz, és add hozzá a saját szavaidat!**

```
      String[] wordListOne = {"24/7","multi-
Tier","30,000 foot","B-to-B","win-win","front-
end", "web-based","pervasive", "smart", "six-
sigma","critical-path", "dynamic"};

   String[] wordListTwo = {"empowered", "sticky",
"value-added", "oriented", "centric", "distributed",
"clustered", "branded","outside-the-box", "positioned",
"networked", "focused", "leveraged", "aligned",
"targeted", "shared", "cooperative", "accelerated"};

   String[] wordListThree = {"process", "tipping-
point", "solution", "architecture", "core competency",
"strategy", "mindshare", "portal", "space", "vision",
"paradigm", "mission"};
```

2 **// Derítsd ki, hány szó, illetve kifejezés van az egyes halmazokban!**

```
   int oneLength = wordListOne.length;
   int twoLength = wordListTwo.length;
   int threeLength = wordListThree.length;
```

3 **// Állíts elő három véletlenszámot!**

```
   int rand1 = (int) (Math.random() * oneLength);
   int rand2 = (int) (Math.random() * twoLength);
   int rand3 = (int) (Math.random() * threeLength);
```

4 **// Most építs fel egy szófordulatot!**

```
   String phrase = wordListOne[rand1] + " " +
wordListTwo[rand2] + " " + wordListThree[rand3];
```

5 **// Írd ki a blablát!**

```
   System.out.println("What we need is a " + phrase);
   }
}
```

Frázisautomata

Hogyan működik?

Dióhéjban: a program létrehoz három kifejezéslistát, majd véletlenszerűen kiválaszt egy-egy szót, illetve kifejezést a három listából, és kiírja az eredményt. Ne aggódj, ha nem érted *pontosan*, hogy mi történik az egyes sorokban. Isten szerelmére, hiszen még az egész könyv előtted áll! Lazíts! Most csak egy gyors pillantást vetünk egy 30 kilométeres, formabontó, célzott, tőkeáttételes paradigmára.

1. Az első lépés a három karakterlánctömb – a szavakat, illetve kifejezéseket tartalmazó tárolók – létrehozása. Egy tömböt (array) bevezetni és létrehozni könnyű. Íme egy aprócska tömb:

```
String[] hazikedvencek = {"Fido", "Zeus", "Bin"};
```

Minden szót idézőjelbe kell tenned (mint minden jó karakterláncot), és vesszővel kell elválasztanod őket.

2. Az a cél, hogy véletlenszerűen kiválassz egy-egy szót mindhárom listából (tömbből), ezért tudnod kell, hogy hány szó található az egyes listákban. Ha valamelyik lista 14 szót tartalmaz, akkor egy 0 és 13 közötti véletlenszámra lesz szükséged (a Java-tömbök elemeinek sorszámozása nullától indul, tehát az első szó helyét a 0, a másodikét az 1, a 14 elemű lista utolsó szavát pedig a 13 jelöli). Szerencsére a Java-tömbök boldogan elárulják neked a hosszukat, csak meg kell kérdezned. A házikedvencek tömbjének esetében ezt írhatod:

```
int x = pets.length;
```

Az **x** értéke ebben az esetben 3.

3. Három véletlenszámra van szükség. A Java becsomagolva, zsugorfóliázva, a polcról leemelve, alapkompetenciaként tartalmaz egy csomó matematikai függvényt (pontosabban tagfüggvényt, de ez most nem érdekes). A **`random()`** tagfüggvény egy 0 és „majdnem 1" közötti véletlenszámot ad vissza, amit meg kell szoroznod az adott listában található elemek számával (a tömb hosszával). Az eredménynek mindenképpen egész számnak kell lennie (tizedesjegyekről szó sem lehet!), ezért a kódban egy típusátalakítás is szerepel (ezt részletesen majd a 4. fejezetben tárgyaljuk). Ugyanazt kell csinálnod, mint bármely más lebegőpontos (float) szám esetében, amit egésszé szeretnél alakítani:

```
int x = (int) 24.6;
```

4. A következő lépés a szófordulat felépítése a három listából kiválasztott szavak összetapasztásával (illetve a szavak közé egy-egy szóköz beszúrásával). Erre a + műveletet használhatod, amely *összefűzi* a String típusú objektumokat. Ahhoz, hogy kiolvass egy elemet egy tömbből, az elem sorszámát (helyét) kell megadnod a tömb neve után:

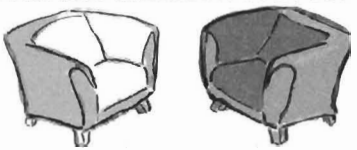
```
String s = pets[0];  // az s értéke most a „Fido" karakterlánc
s = s + " " + "egy kutya";  // az s értéke most ez: „Fido egy kutya"
```

5. Végül kiírod a frázist a parancssorba, és voilá! – máris marketinges vagy!

Amire szükségünk van, az egy...

- átütő célzott folyamat
- dinamikus formabontó fordulópont
- intelligens elosztott alapkompetencia
- éjjel-nappali önvezérlő márkaismeret
- 30 kilométeres kölcsönösen előnyös jövőkép
- hat szigmás hálózatos portál

KANDALLÓ MELLETTI CSEVELY



Ma esti vendégeink, a fordítóprogram és a JVM, azon vitatkoznak, hogy melyikük a fontosabb

JVM

Ez kérdés? Maga viccel! ***Helló!*** *Én* vagyok a Java! Én vagyok az a fickó, aki *futtatja* a programokat. A fordító csak egy *fájlt* ad. Ennyi. Csak egy fájlt. Kinyomtathatja, felragaszthatja a falra tapétának, meggyújthatja, beteheti egy kalitkába, hogy legyen mire potyogtatnia a madaraknak, vagy *bármi*, de egy fájl nem *csinál* semmit, amíg én nem futtatom.

Na, ez a másik dolog: a fordítónak nincs humorérzéke. Persze ha *Önnek* kellene egész nap apró-cseprő nyelvtani hibákat vadásznia…

Én nem mondtam, hogy maga *teljesen* haszontalan, de most komolyan: mit is csinál? Tényleg fogalmam nincs róla. A programozó megírhatná saját kezűleg a bájtkódot, és én ugyanúgy megenném. Hamarosan nem lesz munkád, öreg.

Jó, igyekszem én is karót nyelt lenni, de még mindig nem válaszolt a kérdésemre: mit *csinál* Ön valójában?

Fordító

Nem tetszik ez a hangnem.

Már bocsánat, de *nélkülem* mit is futtatna Ön? Csak a maga okulására: *okkal* tervezték meg úgy a Javát, hogy bájtkódfordítót alkalmazzon. Ha a Java tisztán értelmező nyelv lenne, és a Java-kódokat – futáskor – a virtuális gépnek kellene lefordítania az egyenesen a szövegszerkesztőből kipottyant forráskódból, a Java-programok olyan nevetségesen lassan vánszorognának, mint egy gleccser. Éppen elég időbe telt, amíg a Java meggyőzte az embereket, hogy végre kellően gyors és hatékony a legtöbb feladathoz.

Már elnézést, de ez elég tudatlan (arról nem is beszélve, hogy mennyire *arrogáns*) nézőpont. *Elméletileg igaz*, hogy Ön bármilyen megfelelően formázott bájtkódot tudna futtatni, nem csak olyat, amit egy Java-fordítótól kap, de a gyakorlatban ez képtelenség. Ha a programozó saját kezűleg írná meg a bájtkódot, az olyan lenne, mintha a szövegszerkesztés abból állna, hogy nyers PostScript-kódot írunk. És örülnék neki, ha *nem* „öregezne" le.

JVM

De hát egyes hibák így is eljutnak hozzám! Jelezhetek ClassCastException kivételt, és olyan is előfordul, hogy valaki más típusú dolgot próbál egy tömbbe tenni, mint amit elvileg tárolhatna benne, sőt…

Jó, rendben. De mi a helyzet a *biztonsággal*? Nézze meg, mennyi biztonsági intézkedést hajtok végre én – maga meg *pontosvesszőket* vagy miket ellenőriz? Húha, ez aztán nagy biztonsági kockázat! Még jó, hogy maga itt van!

Legyen, ahogy akarja. De ugyanezeket *nekem is* el kell végeznem, csak hogy tutira ne lopózzon be senki maga után, és változtassa meg a bájtkódot futtatás előtt.

Számíthatsz rá, *öreg*.

Fordító

Ne feledje, hogy a Java erősen típusos nyelv, ami azt jelenti, hogy nem engedhetem meg, hogy egy változóban helytelen típusú adat legyen. Ez létfontosságú biztonsági intézkedés, és én képes vagyok kiküszöbölni a hibák túlnyomó többségét, mielőtt a kód a maga kezébe jutna. Ezenkívül…

Bocsánat, még nem fejeztem be. Igen, *vannak* bizonyos adattípus-kivételek, amelyek futásidőben jelentkezhetnek, de ezek némelyikét azért kell megengedni, hogy lehetővé tegyük a dinamikus kötést, ami szintén fontos szolgáltatása a Javának. Futás közben a Java-programok olyan új objektumokat is betölthetnek, amelyekről a programozó még nem *tudhatott*, ezért némileg rugalmasnak kell lennem. A feladatom az, hogy megakadályozzak minden olyasmit, ami *soha* nem sikerülhet futásidőben. Általában meg tudom mondani, ha valami nem fog működni – például ha a programozó véletlenül egy Button objektumot próbál Socket kapcsolatként használni, azt észreveszem, és meg tudom védeni attól, hogy futáskor kárt tegyen.

Bocsásson meg, de ahogy mondják, én vagyok a védelem első vonala. Az adattípus-megsértések, amelyeket korábban leírtam, tönkretennék a programokat, ha nem küszöbölném ki őket. Ezenkívül én vagyok az, aki megakadályozza a jogosulatlan hozzáférést, például amikor egy kódrész egy privát tagfüggvényt próbál meghívni, vagy egy olyan tagfüggvényt próbál megváltoztatni, ami – biztonsági okokból – nem módosítható. Én akadályozom meg, hogy valaki olyan kódhoz nyúljon, amit nem szabad látnia, beleértve azt is, ha egy kódrész egy másik osztály létfontosságú adataihoz próbál hozzáférni. A munkám fontosságát órákba vagy talán napokba telne leírni.

Természetesen, de ahogy korábban jeleztem, ha én nem akadályoznám meg a lehetséges problémák 99%-át, maga rögtön lefagyna.
De azt hiszem, lejárt az időnk. Majd legközelebb folytatjuk ennek a megvitatását.



Gyakorlat

Kódmágnesek



Egy működőképes Java-program kódja összekeveredett a hűtőmágneseken. Össze tudod rakni úgy a részleteket, hogy előálljon belőle a lap alján látható kimenetet eredményező Java-program? Néhány kapcsos zárójel leesett a padlóra, és túl kicsi volt ahhoz, hogy fel tudjuk venni, úgyhogy nyugodtan egészítsd ki a kódot annyi kapcsos zárójellel, amennyit csak szükségesnek érzel!

```
if (x == 1) {
    System.out.print("d");
    x = x - 1;
}
```

```
if (x == 2) {
    System.out.print("b c");
}
```

```
class Shuffle1 {
  public static void main(String [] args) {
```

```
if (x > 2) {
    System.out.print("a");
}
```

```
int x = 3;
```

```
x = x - 1;
System.out.print("-");
```

```
while (x > 0) {
```

Kimenet:

```
File Edit Window Help Sleep
% java Shuffle1
a-b c-d
```



Gyakorlat

Légy a fordító!



Az ezen az oldalon található Java-fájlok mindegyike egy-egy teljes forrásfájl. A te feladatod az, hogy eljátszd a fordítóprogram szerepét, és kiderítsd, hogy lefordíthatók-e ezek a fájlok. Ha nem, hogyan javítanád ki őket?

A

```
class Exercise1b {
  public static void main(String [] args) {
    int x = 1;
    while ( x < 10 ) {
      if ( x > 3) {
        System.out.println("big x");
      }
    }
  }
}
```

B

```
public static void main(String [] args) {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("small x");
      }
    }
}
```

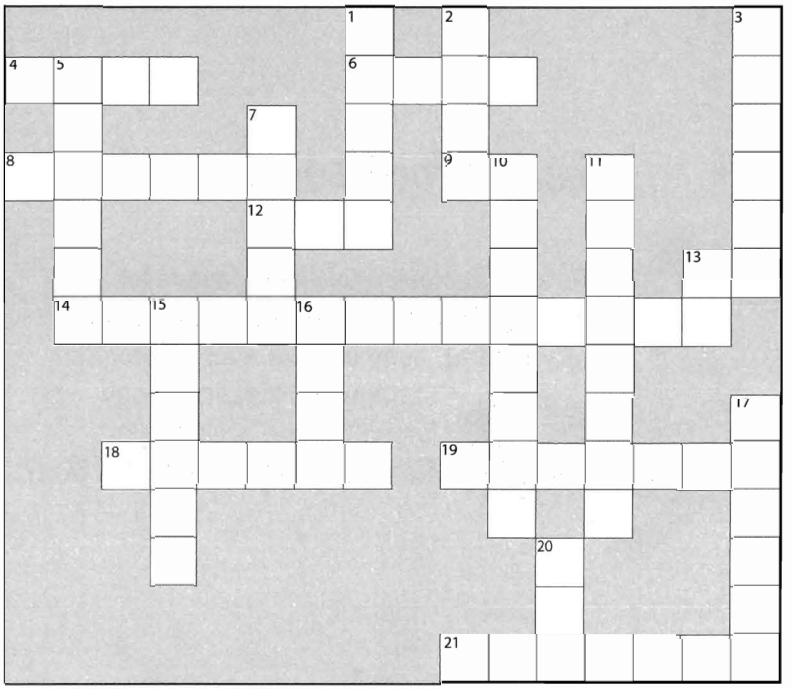
C

```
class Exercise1b {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("small x");
      }
    }
}
```



Java-keresztrejtvény 7.0

Adj munkát az agyadnak!
Ez egy szokványos keresztrejtvény, de a megoldás szinte minden szava az 1. fejezetből való. Azért nem mindegyik, mert bedobtunk néhány (nem a Javával kapcsolatos) műszaki kifejezést is, csak hogy ébren tartsunk.



Vízszintes

4. Parancssori indító utasítás a Java-programokhoz
6. Ismétlődő kódrész (angolul)
8. Elágazás (angolul)
9. Az egyenáram angol rövidítése
12. Számváltozó-típus
13. Az integrált áramkör angol rövidítése
14. Kiíró függvény
18. Karakterek sorozata
19. Új osztályt vagy tagfüggvényt bevezetni (angolul)
21. Amit a készenléti jel után beírsz (angolul)

Függőleges

1. Nem egész (angolul)
2. Üres kézzel visszatérő
3. Nyilvános, angolul
5. „Dolgokat" tárolhatsz bennük (többesszámban, angolul)
7. Amíg
10. A forráskóddal táplálkozó program (angolul)
11. Nem állandó (angolul)
13. A LAN-zsokék részlege
15. Sokkoló módosító
16. Egynek muszáj lennie belőle
17. Ezzel végezhetsz el egy feladatot
20. Ő dolgozza fel a bájtkódot



Összekevert ÜZENETEK

Nézd meg az alábbi rövid Java-programot. A program egyik kódblokkja hiányzik. A feladatod az, hogy **összepárosítsd a bal oldali kódjelölteket** azzal a **kimenettel**, amit akkor látnál, ha az adott blokkot szúrnád be a kódba. Nem minden kimenetsornak van párja, viszont egyes kimenetsorok többször is felhasználhatók. Kösd össze vonalakkal a kódjelölteket és a nekik megfelelő parancssori kimeneteket. (A megoldást a fejezet végén találod.)

```
class Test {
  public static void main(String [] args) {
    int x = 0;
    int y = 0;
    while ( x < 5 ) {
      [                                    ]
      System.out.print(x + "" + y +" ");
      x = x + 1;
    }
  }
}
```

Ide jönnek a kódjelöltek

Párosítsd össze az egyes jelölteket egy-egy lehetséges kimenettel

Kódjelöltek:

```
y = x - y;
```

```
y = y + x;
```

```
y = y + 2;
if( y > 4 ) {
  y = y - 1;
}
```

```
x = x + 1;
y = y + x;
```

```
if ( y < 5 ) {
  x = x + 1;
  if ( y < 3 ) {
    x = x - 1;
  }
}
y = y + 2;
```

Lehetséges kimenetek:

```
22 46
```

```
11 34 59
```

```
02 14 26 38
```

```
02 14 36 48
```

```
00 11 21 32 42
```

```
11 21 32 42 53
```

```
00 11 23 36 410
```

```
02 14 25 36 47
```



Halászd ki a medencéből!



A ***feladatod*** az, hogy kihalászd a kóddarabokat a medencéből, és elhelyezd azokat a kód üresen hagyott helyein. Egy kóddarabot **nem** használhatsz fel egynél többször, viszont nem lesz szükséged minden darabra. A ***cél*** egy olyan osztály felépítése, amely lefordítható, futtatható, és az alább látható kimenetet eredményezi. Vigyázat – ez a feladat nehezebb, mint gondolnád!

Kimenet

```
File  Edit  Window  Help  Cheat
%java PoolPuzzleOne
a noise
annoys
an oyster
```

```
class PoolPuzzleOne {
  public static void main(String [] args) {
    int x = 0;

    while ( __________ ) {

      _____________________________
      if ( x < 1 ) {
        _____________________________
      }
      _____________________________

      if ( __________ ) {

        _____________________________

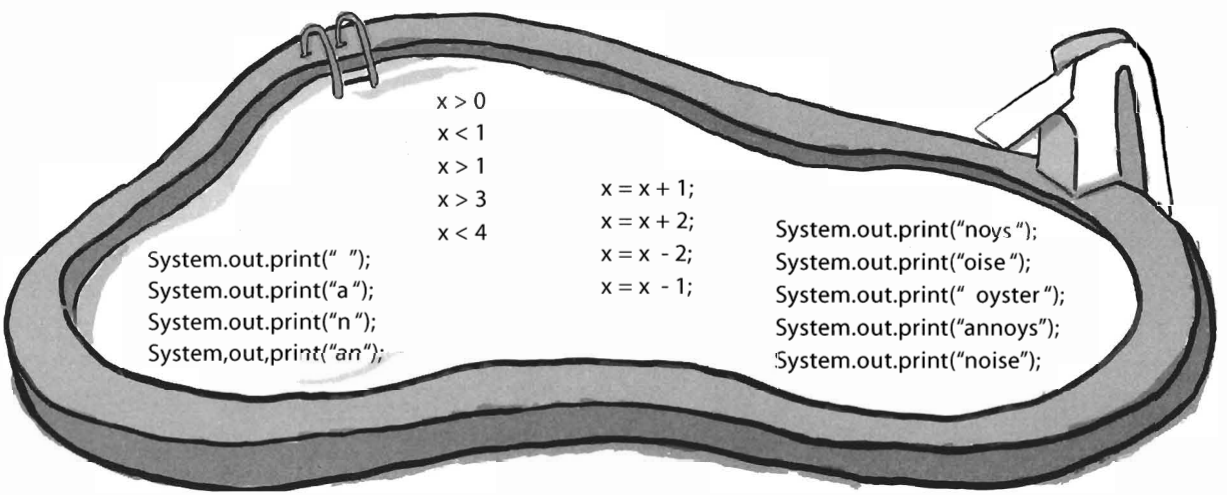
        __________
      }
      if ( x == 1 ) {

        _____________________________
      }
      if ( __________ ) {

        _____________________________
      }
      System.out.println("");

      __________
    }
  }
}
```

Megjegyzés: A medencében úszó kóddarabok mindegyikét csak egyszer használhatod fel!





Gyakorlat – Megoldás

Kódmágnesek:

```
class Shuffle1 {
  public static void main(String [] args) {

    int x = 3;
    while (x > 0) {

      if (x > 2) {
        System.out.print("a");
      }

      x = x - 1;
      System.out.print("-");

      if (x == 2) {
        System.out.print("b c");
      }

      if (x == 1) {
        System.out.print("d");
        x = x - 1;
      }
    }
  }
}
```

```
File Edit Window Help Poet
% java Shuffle1
a-b c-d
```

A

```
class Exercise1b {
  public static void main(String [] args) {
    int x = 1;
    while ( x < 10 ) {
      x = x + 1;
      if ( x > 3) {
        System.out.println("big x");
      }
    }
  }
}
```

Ez lefordítható és futtatható (kimenetet nem eredményez), de a kiemelt sort még hozzá kell adni, különben a program végtelen „while" ciklusban ragad, és örökké futni fog!

B

```
class Foo {
  public static void main(String [] args) {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("small x");
      }
    }
  }
}
```

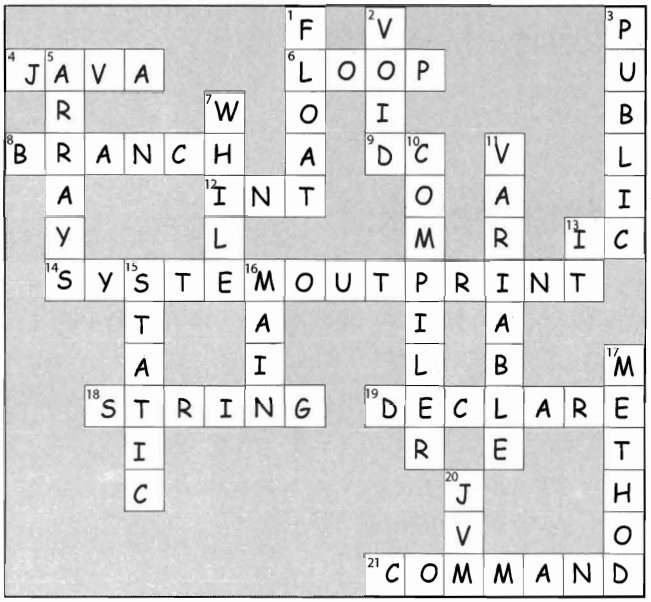
Ez a fájl egy osztály bevezetése nélkül nem fordítható le. Ne feledkezz meg a záró kapcsos zárójelről sem!

C

```
class Exercise1b {
  public static void main(String [] args) {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("small x");
      }
    }
  }
}
```

A „while" ciklus kódjának egy tagfüggvényen belül kell lennie, nem lófrálhat csak úgy az osztályban.






```
class PoolPuzzleOne {
  public static void main(String [] args) {
    int x = 0;

    while ( X < 4 ) {

      System.out.print("a");
      if ( x < 1 ) {
        System.out.print(" ");
      }
      System.out.print("n");

      if ( X > 1 ) {

        System.out.print(" oyster");
        x = x + 2;
      }
      if ( x == 1 ) {

        System.out.print("noys");
      }
      if ( X < 1 ) {

        System.out.print("oise");
      }
      System.out.println("");

      X = X + 1;
    }
  }
}
```

```
File Edit Window Help Cheat
%java PoolPuzzleOne
a noise
annoys
an oyster
```

```
class Test {
  public static void main(String [] args) {
    int x = 0;
    int y = 0;
    while ( x < 5 ) {
      [          ]

      System.out.print(x + "" + y +" ");
      x = x + 1;
    }
  }
}
```

Kódjelöltek:

```
y = x - y;
```

```
y = y + x;
```

```
y = y + 2;
if( y > 4 ) {
  y = y - 1;
}
```

```
x = x + 1;
y = y + x;
```

```
if ( y < 5 ) {
  x = x + 1;
  if ( y < 3 ) {
    x = x - 1;
  }
}
y = y + 2;
```

Lehetséges kimenetek:

```
22 46
```

```
11 34 59
```

```
02 14 26 38
```

```
02 14 36 48
```

```
00 11 21 32 42
```

```
11 21 32 42 53
```

```
00 11 23 36 410
```

```
02 14 25 36 47
```

Utazás Objektumfalvára



Azt mondtátok, hogy objektumokkal fogok találkozni. Az 1. fejezetben minden kódot a main() tagfüggvénybe tettünk, ami nem igazán objektumközpontú dolog. Sőt, *egyáltalán* nem az. *Használtunk* persze néhány objektumot, például a String tömböket a Frázisautomatában, de semmilyen saját objektum*típust* nem dolgoztunk ki. Most azonban végre magunk mögött hagyjuk az eljárások világát, elhúzunk a main()-ből, és nekilátunk saját objektumokat készíteni. Látni fogod, mi teszi a Java nyelvű objektumközpontú („objektumorientált", OO) fejlesztést olyan szórakoztatóvá. Megtanulsz különbséget tenni *osztály* és *objektum* között. Megtudod, miként tehetik az objektumok jobbá az életedet (legalábbis ami a programozást illeti – a ruhatáradon sajnos nem segíthetünk). Vigyázat: ha egyszer beteszed a lábad Objektumfalvára, nem biztos, hogy valaha visszamész! Ha így lenne, küldj nekünk képeslapot!

Székek háborúja

(avagy: Hogyan változtatják meg az objektumok az életedet?)

Egyszer volt, hol nem volt, volt egyszer egy szoftverműhely, ahol két programozó ugyanazt a tervet kapta, hogy „építse fel". A Nagyon Idegesítő Főnök versenyre kényszerítette a két fejlesztőt, azt ígérve nekik, hogy aki előbb teszi le az asztalra a kész programot, az kap egy király Aeron™ széket – olyat, amilyen a Szilícium-völgyi srácoknak van. Larry, az eljárásokra esküdő programozó, és Brad, az objektumok megszállotja egyaránt fellelkesült a kilátásba helyezett díjtól.



Larry az irodájában ezen morfondírozott: „Mit kell *csinálnia* ennek a programnak? Milyen ***eljárásokra*** van szükség?". Meg is adta magának a választ: „**rotate** (forgat) és **playSound** (lejátszikHang)". Nekilátott hát megírni az eljárásokat. Végtére is, *mi* egy program, ha nem egy kupac eljárás?

Brad közben egy kávézóban lazított, és ezen járt az agya: „Milyen ***dolgok*** vannak ebben a programban? Kik a kulcs*játékosok*?" Először is – gondolta – vannak az **Alakzatok** (Shape). Persze más objektumok – a Felhasználó, a Hang és a Kattintás esemény – is az eszébe jutottak, de ezekhez már volt egy kódkönyvtára, ezért az Alakzatok felépítésére összpontosított. Olvass tovább, hogy megtudd, hogyan építette fel Brad és Larry a saját programját, és választ kapj az égető kérdésre: ***„Ki kapta az Aeront?"***.



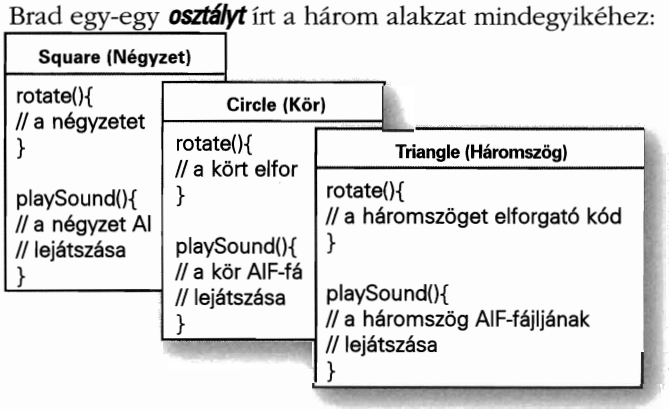
Larry irodájában

Ahogy korábban már kismilliószor, Larry nekilátott megírni a **Fontos Eljárásait**. A **rotate** és a **playSound** hipp-hopp el is készült:

```
rotate(shapeNum) {
  // az alakzat elforgatása 360°-kal
}
playSound(shapeNum) {
  // a shapeNum adja meg, melyik
  // AIF-fájlt kell lejátszani
}
```

Brad laptopján a kávézóban

Brad egy-egy ***osztályt*** írt a három alakzat mindegyikéhez:



Larry azt hitte, megvan. Szinte érezte az Aeron csodás acélvázát a... De várjunk csak! Változott a terv!

„Na jó, *tulajdonképpen* te voltál az első, Larry," – mondja a Főnök – „de még egy aprósággal ki kell egészíteni a programot. Ez persze nem okozhat gondot két olyan klassz programozónak, mint ti."

„Ha annyi ezresem lenne, ahányszor ezt hallottam..." – gondolja Larry, tudván, hogy problémamentes tervmódosítás nem létezik. *„Brad viszont meglepően nyugodt. Vajon miért?"* Larry azért nem izgul, mert mélyen hisz benne, hogy az objektumközpontú megoldások – bár jópofák – egyszerűen túl lassúak. Csak úgy győzheted meg az ellenkezőjéről, ha kioperálod és átprogramozod az agyát.



← **a terv kiegészítése**

Ismét Larry irodájában

A rotate eljárás továbbra is működhet, mert a kód egy keresőtáblát használ a shapeNum (az alakzat sorszáma) és a hozzá tartozó alakzatgrafika összepárosítására. A ***playSound eljáráson azonban változtatni kell.*** És mi a fene az a .hif fájl?

```
playSound(shapeNum) {
   // ha az alakzat nem amőba,
      // a shapeNum-mal kikeressük a leját-
      // szandó AIF hangot, és lejátsszuk
   // else
      // lejátsszuk az amőba .hif hangfájlját
  }
```

Végül is, nem nagy ügy, de Larry ***fanyalogva nyúl minden korábban már letesztelt kódhoz.*** *Mindenki* közül *ő* tudja a legjobban, hogy nem számít, mint mond a főnök, a ***terv mindig változik***.

Brad laptopján a strandon

Brad csak elmosolyodik, beleszürcsöl a Margaritájába, és *ír egy új osztályt.* Az objektumközpontú megközelítésben azt szereti a legjobban, hogy nem kell hozzányúlnia a már ellenőrzött és leadott kódokhoz. „Rugalmasság, bővíthetőség..." – gondolja – „az objektumközpontúság előnyei".



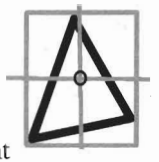
Larry néhány pillanattal megelőzi Bradet.

(Haha! Ennyit az objektumok felsőbbrendűségéről...) Larry arcáról azonban lefagy a mosoly, amikor a Nagyon Idegesítő Főnök (csalódott hangon) azt mondja: „Jaj, ne. Az amőbának *nem így* kellene forognia..."

Amint kiderül, mindkét programozó így írta meg a forgató kódot:

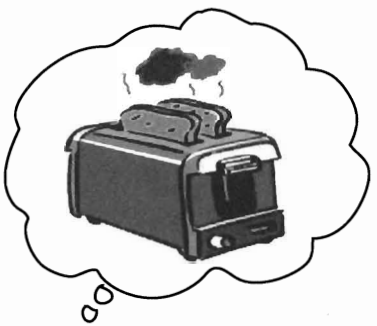
1) Meghatározzuk az alakzatot körülvevő négyszöget.

2) Kiszámítjuk ennek a négyszögnek a középpontját, és az alakzatot e körül a pont körül forgatjuk el.

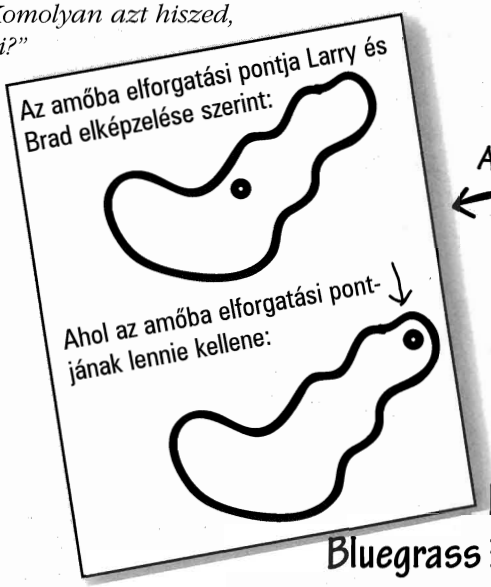


Az amőba alakzatnak azonban csak az egyik *vége* kellene, hogy a pont körül forogjon, úgy, mint egy óramutató.

„Leégtem." –gondolja Larry, és a lelki szemei előtt megjelenik egy odaégett pirítós – „Habár... hozzáadhatnék egy újabb if/else részt a rotate eljáráshoz, aztán egyszerűen bedrótozhatnám az amőba elforgatási középpontjának kódját. Ezzel valószínűleg nem tennék tönkre semmit." Egy hang a fejében azonban azt súgja: *„Szarvashiba. Komolyan azt hiszed, hogy a terv nem fog többet változni?"*



Ismét Larry irodájában



Amit a terv elfelejtett közölni

Brad laptopján a Telluride Bluegrass Festivalon, a fűben ücsörögve

Larry arra a következtetésre jut, hogy jobban jár, ha elforgatásipont-argumentumokat ad a rotate eljáráshoz. Ez ***egy csomó kódot érint.*** Tesztelés, újrafordítás, az egész cucc elölről. Ami korábban működött, annak annyi.

```
rotate(shapeNum, xPt, yPt) {
 //  ha az alakzat nem amőba,
    //  kiszámítjuk a középpontot
    //  egy négyszög alapján,
    //  aztán forgatunk
 // else
    //  fogjuk az xPt-t és az yPt-t,
    //  mint az elforgatási pont eltolását,
    //  aztán forgatunk
}
```

Brad nem esik ki a ritmusból: módosítja az elforgató **tagfüggvényt**, de csak az Amoeba osztályban. A program többi részének ***már letesztelt, működő, lefordított kódjához nem kell hozzányúlnia***.

Az amőba elforgatási pontját egy **jellemzővel** (attribute) adja meg, amellyel minden amőba objektum rendelkezni fog. Az új programváltozatot egyetlen Bela Fleck-műsor alatt módosítja, ellenőrzi és (drót nélküli kapcsolaton át) leadja.

Amoeba

```
int xPoint
int yPoint

rotate(){
// az amőbát az x és az y
// alapján elforgató kód
}

playSound(){
// az amőba új .hif-fájlját
// lejátszó kód
}
```

Szóval az objektumos Brad kapja a széket, igaz?

Ne siessünk ennyire. Larry ugyanis felfedez egy hibát Brad megoldásában. Mivel pedig biztos benne, hogy a szék birtokában Lucy kegyeit is elnyerné a könyvelési osztályról, muszáj fordítania a szerencséjén.

LARRY: Ismétlődik a kódod! A rotate eljárás mind a négy alakzatizében benne van!

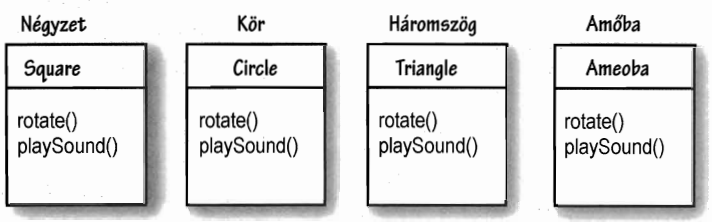
BRAD: ***Tagfüggvény***, nem *eljárás*. És ezek ***osztályok***, nem *izék*.

LARRY: Nem mindegy? Ez egy hülye szerkezet: *négy* különböző rotate „tagfüggvényt" kell fenntartanod! Mi a jó ebben?

BRAD: Ó, ezek szerint nem láttad a végső tervet. Gyere Larry, megmutatom, hogy működik az objektumközpontú **öröklés**!



Ezt akarja Larry (és biztos benne, hogy a szék lenyűgözné a lányt)

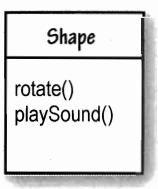


1

Megnéztem, mi a közös a négy osztályban.

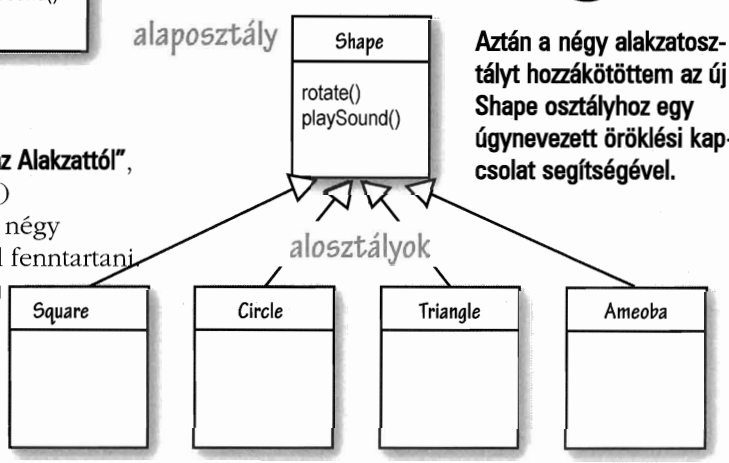
2

Mindegyik alakzat (Shape), és mind forog (rotate), illetve hangot játszik le (playSound). Így hát elvonttá tettem ezeket a közös vonásokat, és egy új, Shape nevű osztályba tettem őket.



3

Aztán a négy alakzatosztályt hozzákötöttem az új Shape osztályhoz egy úgynevezett öröklési kapcsolat segítségével.



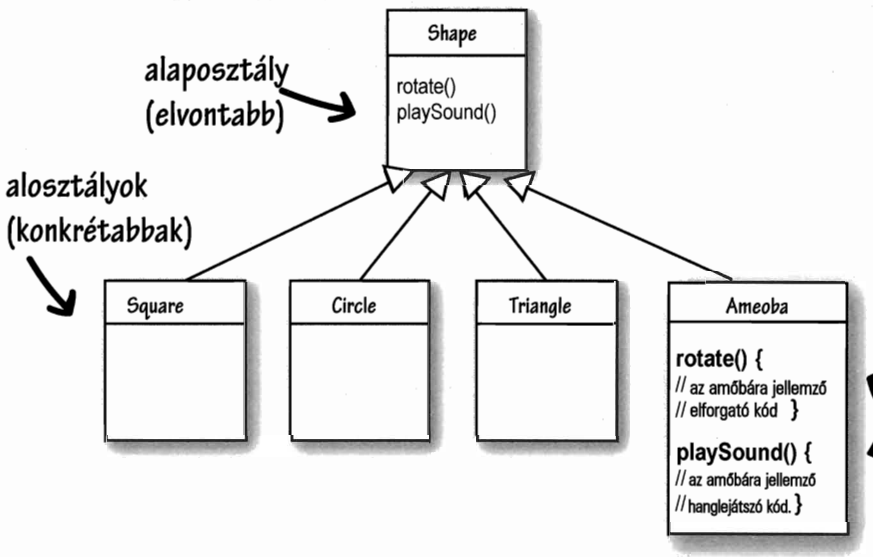
Ezeket úgy lehet olvasni, hogy **„a Négyzet örököl az Alakzattól"**, **„a Kör örököl az Alakzattól"**, és így tovább. A rotate() és playSound() tagfüggvényeket eltávolítottam a négy alakzatosztályból, így már csak egy példányt kell fenntartani. A Shape osztály a négy másik osztály **alaposztálya** (ősosztálya, szuperosztálya; superclass), azok pedig **alosztályai** (subclass) a Shape-nek. Az alosztályok öröklik az alaposztályuk tagfüggvényeit – más szavakkal, ha a Shape osztály rendelkezik valamilyen vonással, akkor az az alosztályokban is automatikusan megtalálható lesz.

Mi a helyzet az amőba forgatásával?

LARRY: De hát nem az volt a probléma, hogy az amőba alakzatnak teljesen más forgató és hanglejátszó eljárásra volt szüksége?
BRAD: Tagfüggvényre.
LARRY: Tökmindegy. Hogyan csinálhat mást az amőba, ha a viselkedését a Shape osztálytól „örökli"?



BRAD: Ez az utolsó lépés. Az Amoeba osztály **felülbírálja** (override) a Shape osztály tagfüggvényeit. Így futáskor a JVM pontosan tudja, hogy melyik rotate() tagfüggvényt kell elindítania, amikor azt az utasítást kapja, hogy forgassa el az amőbát.



4

Az Amoeba osztályban felülbíráltam a Shape alaposztály rotate() és playSound() tagfüggvényeit.

A felülbírálás (felülírás) csak annyit jelent, hogy egy alosztály újrafogalmazza az egyik örökölt tagfüggvényét, ha módosítania vagy bővítenie kell annak viselkedését.

LARRY: És hogyan „utasítasz" egy amőbát, hogy csináljon valamit? Nem kell meghívnod az eljárást – bocsánat, *tagfüggvényt* –, hogy aztán megmondd neki, *melyik* izét forgassa el?

BRAD: Ez az igazán klassz az objektumközpontúságban. Amikor mondjuk el kell forgatni a háromszöget, a programkód hívja meg a rotate() tagfüggvényt a *Triangle objekumra*. A program többi részének nem is kell tudnia, *hogyan* forog a háromszög, és nem is érdekli őket. Ha pedig valamilyen újdonsággal kell kiegészítened a programot, csak írsz egy új osztályt az új objektumtípus számára, így **az új objektumoknak saját viselkedésük lesz**.



Megöl a kíváncsiság.

Ki nyerte a széket?



Amy a második emeletről.
(Ugyanis a Főnök – mindenki előtt titokban tartva – *három* programozónak adta oda a tervet.)

Mit szeretsz az objektumközpontú programozásban?

„Természetesebben tervezhetek, és az elemek tovább fejlődhetnek."

– Joy (27), szoftvertervező

„Nem kell belenyúlnom olyan kódokba, amiket már leteszteltem, ha új szolgáltatást szeretnék adni egy programhoz."

– Brad (32), programozó

„Tetszik, hogy az adatok és a rajtuk műveleteket végző tagfüggvények együtt, egyetlen osztályban vannak."

– Josh (22), sörivó

„A kódok újrahasznosításának lehetőségét más alkalmazásokban. Amikor egy új osztályt írok, elég rugalmassá tehetem ahhoz, hogy később máshol is felhasználható legyen."

– Chris (39), projektvezető

„Nem hiszem el, hogy Chris ezt mondta. 5 éve nem írt egyetlen sor kódot sem."

– Daryl (44), Chris beosztottja

„A széken kívül?"

– Amy (34), programozó

Agytorna

Ideje, hogy megdolgoztasd az agysejtjeidet.
Az imént olvashattál egy történetet egy eljárásokkal dolgozó programozóról, aki versenyre kelt egy objektumközpontú fejlesztővel. Rövid áttekintést kaptál az objektumközpontú programozás néhány kulcsfontosságú fogalmáról: az osztályokról, tagfüggvényekről és jellemzőkről. A fejezet hátralevő részében osztályokat és objektumokat fogunk vizsgálgatni (az öröklésre és a felülbírálásra a későbbi fejezetekben még visszatérünk). Az eddig látottakat – és azt, amit esetleg tudhatsz egy másik objektumközpontú nyelven végzett munkából – alapul véve állj meg egy percre, hogy feltedd magadnak az alábbi kérdéseket: Milyen alapvető dolgokat kell végiggondolnod, amikor megtervezel egy Java-osztályt? Milyen kérdésekre kell választ találnod? Ha írnod kellene egy ellenőrzőlistát, amit felhasználhatsz, amikor osztályokat tervezel, mi lenne a listán?

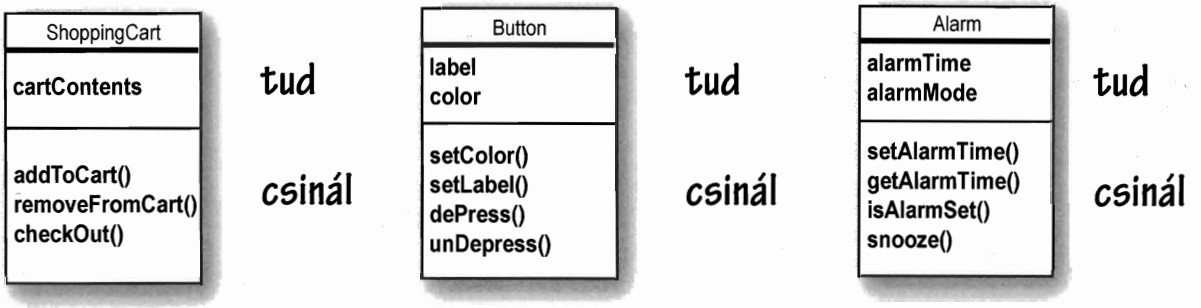


metakognitív tipp

Ha elakadsz egy gyakorlattal, mondd ki hangosan a kérdést. A beszéd (és a hallás) az agyad más részeit kapcsolja be. A legjobb, ha egy másik emberrel beszéled meg a problémákat, de a házikedvencek is megteszik. A mi kutyánk is így tanulta meg a többalakúság használatát

Amikor megtervezel egy osztályt, gondold végig, milyen objektumok jönnek majd létre az adott osztálytípusból. Az alábbiakat kell meghatároznod:

- amit az objektum **tud**
- amit az objektum **csinál**

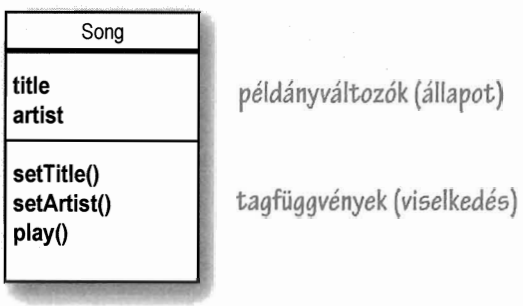


Azokat a dolgokat, amelyeket egy objektum *tud* magáról, úgy hívják, hogy

- példányváltozók (instance variable)

Azokat a dolgokat, amelyeket egy objektum *csinálhat*, úgy hívják, hogy

- tagfüggvények (method)

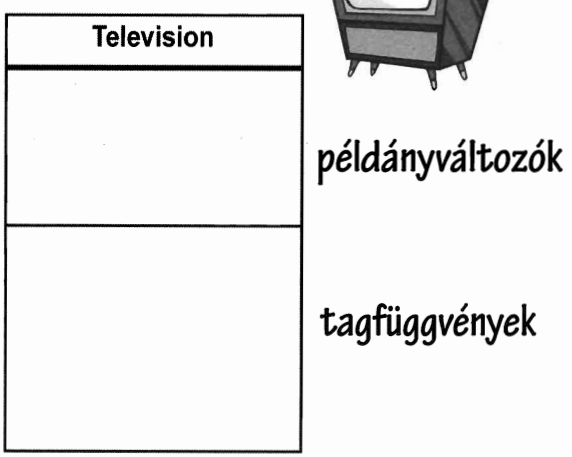


Azokat a dolgokat, amelyeket egy objektum ***tud*** magáról, úgy hívják, hogy **példányváltozók**. Ezek jelképezik az objektum állapotát (az adatokat), és az adott típusba tartozó minden objektum esetében egyedi lehet az értékük.
A **példányra** (instance) gondolhatsz úgy, mintha azt mondanád, hogy **objektum**.
Azokat a dolgokat, amelyeket egy objektum ***csinálhat***, úgy hívják, hogy **tagfüggvény**. Amikor megtervezel egy osztályt, végig kell gondolnod, hogy milyen adatokról kell tudniuk a belőle létrehozott objektumoknak, valamint meg kell határoznod azokat a tagfüggvényeket, amelyek műveleteket végeznek ezeken az adatokon. Az objektumok általában olyan tagfüggvényekkel rendelkeznek, amelyek a példányváltozók értékeit írják vagy olvassák. Az Alarm (Vekker) objektumok például egy példányváltozóban tárolják az ébresztési időt (alarmTime), és van két tagfüggvényük, amelyekkel kiolvashatják és beállíthatják az alarmTime értékét.
Az objektumoknak tehát példányváltozóik és tagfüggvényeik vannak, de ezek az osztály részét képezik.

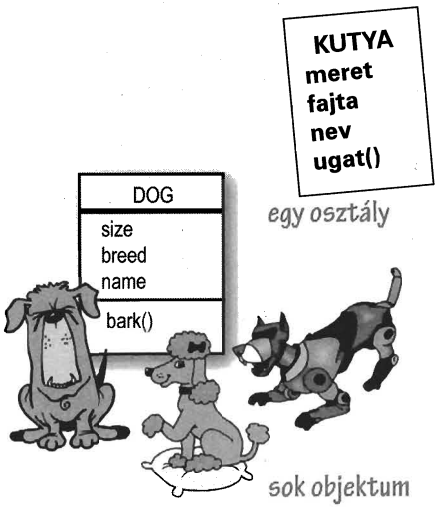
Hegyezd ki a ceruzád!

Írd be, hogy mit kell tudnia és csinálnia egy Television objektumnak:



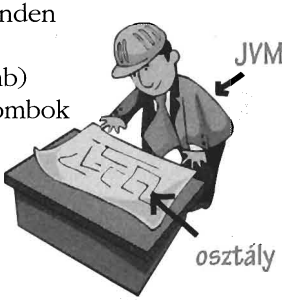


Mi a különbség az osztály és az objektum között?

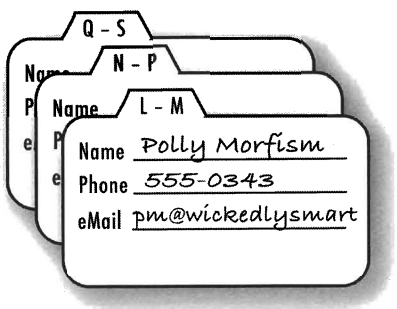


Az osztályok nem objektumok (de az előbbiekből hozzuk létre az utóbbiakat).

Az osztályok az objektumok *tervrajzai*. Az osztály mondja meg a virtuális gépnek, hogyan hozhatja létre az adott típusú objektumot. Az osztályból létrehozott minden objektum saját értékeket ad az osztály példányváltozóinak. Egy Button (Gomb) nevű osztályból például különböző gombok tucatjait hozhatod létre, és mindegyiknek más színe, mérete, alakja, felirata stb. lehet.



Nézd így...



Az objektumok olyanok, mint egy-egy bejegyzés a noteszodban.
Az objektumokat elképzelheted úgy, mint egy csomag üres névjegykártyát. Minden kártyán ugyanazok az üres mezők (a példányváltozók) találhatók. Amikor kitöltesz egy kártyát, egy példányt (objektumot) hozol létre, amelynek az állapotát a kártyán szereplő bejegyzések határozzák meg.
Az osztály tagfüggvényei jelképezik azokat a műveleteket, amelyeket a konkrét kártyákon végrehajthatsz – a Nevjegykartya osztály tagfüggvényei lehetnek például a kiolvasNev(), a modositNev() vagy a beallitNev().
Tehát mindegyik kártya ugyanazokat a dolgokat – kiolvasNev(), modositNev() stb. – *csinálhatja*, de olyan dolgokat *tud*, amelyek csak az adott kártyára jellemzőek.

Az első objektumod

Mit kell tenned hát, hogy létrehozz és használatba végy egy objektumot? Először is, *két* osztályra lesz szükséged. Az egyik osztály határozza meg a használni kívánt objektum típusát (Dog, AlarmClock, Television stb.), a másik pedig arra szolgál, hogy *teszteld* az új osztályt. A *tesztelő* osztály lesz az, ahol a main tagfüggvény helyet kap, és ebben a main() tagfüggvényben hozod létre az új típusú objektumokat, illetve végzel rajtuk műveleteket. A tesztelő osztálynak egyetlen feladata van: *kipróbálni* az új objektumosztály-típus tagfüggvényeit és változóit. Innen kezdve a könyv számos példájában két osztállyal fogsz találkozni. Az egyik lesz a *valódi* osztály – az az osztály, amelynek az objektumait ténylegesen használni szeretnénk –, a másik pedig a *tesztosztály*, amelynek az <*OsztályNeve*>**TestDrive** nevet adjuk. Ha az osztályt például **Bungee**-nek hívják, egy **BungeeTestDrive** nevű osztályra is szükséged lesz. Csak az <*OsztályNeve*>**TestDrive** osztály fog rendelkezni main() tagfüggvénnyel, és az egyetlen feladata az lesz, hogy létrehozza az új típusú (a nem a tesztosztályba tartozó) objektumokat, majd a pont (.) művelettel elérje az új objektumok tagfüggvényeit és változóit. A következő példák mindezt teljesen világossá fogják tenni.

A pont művelet (.)

A pont művelettel érheted el egy objektum állapotát és viselkedését (vagyis a példányváltozóit és tagfüggvényeit).

```
// új objektum létrehozása
Dog d = new Dog();

// a kutyát úgy utasíthatod
// ugatásra, hogy
// a d változó után kiteszed
// a pont műveleti jelet,
// és meghívod a bark()
// tagfüggvényt
d.bark();

// a kutya méretét is
// a pont művelettel
// állíthatod be
d.size = 40;
```

1 Megírod az osztályt

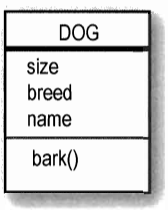
```
class Dog {

  int size;
  String breed;
  String name;

  void bark() {
    System.out.println("Ruff! Ruff!");
  }
}
```

példányváltozók

tagfüggvény



2 Írsz egy tesztosztályt (TestDrive)

ez csak egy main tagfüggvény (kódot majd a következő lépésben teszünk bele)

```
class DogTestDrive {
  public static void main (String[] args) {
    // Ide jön a Dog osztályt tesztelő kód
  }
}
```

3 A tesztosztályban létrehozol egy objektumot, és csinálsz valamit az objektum változóival és tagfüggvényeivel.

```
class DogTestDrive {
  public static void main (String[] args) {
    Dog d = new Dog();
    d.size = 40;
    d.bark();
  }
}
```

Dog objektum létrehozása

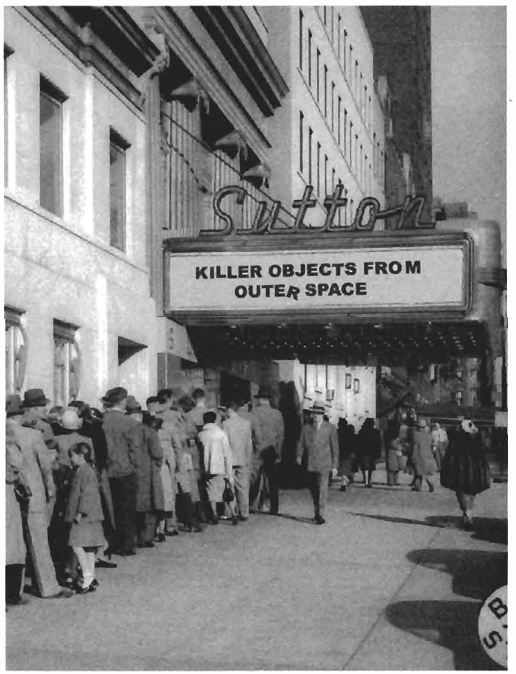
a pont (.) művelettel állíthatod be a Dog méretét

pont művelet

és hívhatod meg az osztály bark() tagfüggvényét

Ha már rendelkezel OO-ismeretekkel, észreveheted, hogy nem alkalmazunk betokozást (egységbe zárást; erről majd a 4. fejezetben lesz szó).

Movie objektumok létrehozása és tesztelése



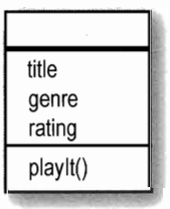

```
class Movie {
  String title;
  String genre;
  int rating;

  void playIt() {
    System.out.println("Playing the movie");
  }
}

public class MovieTestDrive {
  public static void main(String[] args) {
    Movie one = new Movie();
    one.title = "Gone with the Stock";
    one.genre = "Tragic";
    one.rating = -2;
    Movie two = new Movie();
    two.title = "Lost in Cubicle Space";
    two.genre = "Comedy";
    two.rating = 5;
    two.playIt();
    Movie three = new Movie();
    three.title = "Byte Club";
    three.genre = "Tragic but ultimately uplifting";
    three.rating = 127;
  }
}
```

Hegyezd ki a ceruzád!





A MovieTestDrive osztály létrehozza a Movie osztály objektumait (példányait), a pont (.) művelettel értéket ad a példányváltozóknak, valamint meghív egy tagfüggvényt valamelyik objektumra. Töltsd ki a jobb oldalon látható táblázatot azokkal az értékekkel, amelyeket a három objektum a main() végén tartalmazni fog.

1. objektum	title genre rating
2. objektum	title genre rating
3. objektum	title genre rating

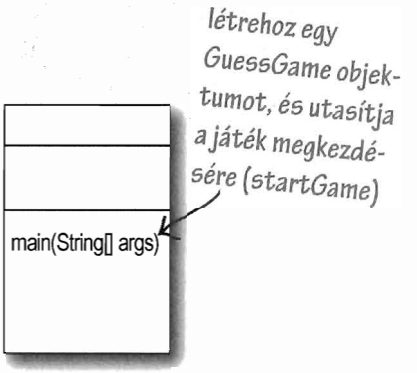
Gyorsan! Kifelé a main-ből!

Amíg a main() tagfüggvényben tartózkodsz, nem igazán teszed be a lábad Objektumfalvára. Egy tesztprogram nyugodtan futhat a main-en belül, de egy valódi objektumközpontú alkalmazásban az objektumoknak társalogniuk kell más objektumokkal, nem úgy, mint a statikus main() tagfüggvényben, amely csak létrehozza és teszteli az objektumokat.

A main két feladata:

- a valódi osztály **tesztelése**
- a Java-alkalmazás **elindítása**

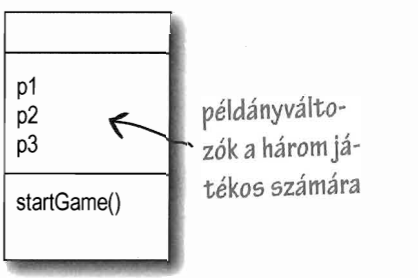
Egy valódi Java-alkalmazás semmi másból nem áll, mint egymással társalgó objektumokból. A *társalgás* ebben az esetben azt jelenti, hogy az objektumok tagfüggvényeket hívnak meg egymásra. Az előző oldalon azt láthattad, hogy egy önálló tesztosztály main() tagfüggvénye segítségével hozzuk létre és próbáljuk ki egy másik osztály tagfüggvényeit és változóit (és erre majd a 4. fejezetben is visszatérünk). A 6. fejezetben viszont majd azt vizsgáljuk, hogyan indíthatsz útjára egy *igazi* Java-alkalmazást egy main() tagfüggvénnyel rendelkező osztályból (objektumok létrehozásával, majd ezeknek az objektumoknak a szabadon engedésével, hogy kapcsolatba léphessenek más objektumokkal).
Lássunk azonban egy aprócska példát, ami ízelítőt nyújt abból, hogy miként viselkednek a valódi Java-alkalmazások! Mivel éppen csak elkezdtük a Java tanulását, az eszköztárunk még szerény, ezért az alábbi programot esetlennek és kevéssé hatékonynak találhatod. Nyugodtan eltöprenghetsz, hogyan tudnál javítani rajta – a későbbi fejezetekben éppen ezt fogjuk tenni. Ne aggódj, ha a kód egyes részeit még zavarosnak találod; ennek a példának csupán annyi a célja, hogy megmutassa, az objektumok más objektumokkal társalognak.



Kitalálós játék

Összefoglalás:

Kitalálós játékunkhoz egy „game" (játék) és három „player" (játékos) objektum szükséges. A játékobjektum egy 0 és 9 közötti véletlenszámot állít elő, a három játékosobjektum pedig megpróbálja kitalálni, melyik szám volt. (Nem állítottuk, hogy különösebben *izgalmas* játék lesz.)

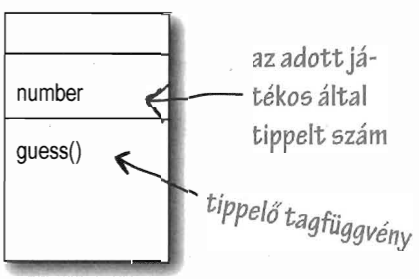


Osztályok:

`GuessGame.class` `Player.class` `GameLauncher.class`

A program logikája:

1) A GameLauncher (játékindító) osztály indítja el az alkalmazást; ebben található a main() tagfüggvény.
2) A main() tagfüggvény létrehoz egy GuessGame (kitalálós játék) objektumot, és meghívja annak startGame() – inditJatek – tagfüggvényét.
3) A GuessGame objektum startGame() tagfüggvénye az a hely, ahol az egész játék lejátszódik. Ez a tagfüggvény létrehozza a három játékost, majd „kigondol" egy véletlenszámot (amit a játékosoknak ki kell találniuk). Ezt követően kér egy-egy tippet a játékosoktól, ellenőrzi a számot, végül pedig vagy kiírja, hogy ki nyert, vagy új tippet kér a játékosoktól.



```
public class GuessGame {
   Player p1;
   Player p2;
   Player p3;

   public void startGame() {
     p1 = new Player();
     p2 = new Player();
     p3 = new Player();

     int guessp1 = 0;
     int guessp2 = 0;
     int guessp3 = 0;

     boolean p1isRight = false;
     boolean p2isRight = false;
     boolean p3isRight = false;

     int targetNumber = (int) (Math.random() * 10);
     System.out.println("I'm thinking of a number between 0 and 9...");

     while(true) {
       System.out.println("Number to guess is " + targetNumber);

       p1.guess();
       p2.guess();
       p3.guess();

       guessp1 = p1.number;
       System.out.println("Player one guessed " + guessp1);

       guessp2 = p2.number;
       System.out.println("Player two guessed " + guessp2);

       guessp3 = p3.number;
       System.out.println("Player three guessed " + guessp3);

       if (guessp1 == targetNumber) {
          p1isRight = true;
       }
       if (guessp2 == targetNumber) {
          p2isRight = true;
       }
       if (guessp3 == targetNumber) {
         p3isRight = true;
       }

       if (p1isRight || p2isRight || p3isRight) {

         System.out.println("We have a winner!");
         System.out.println("Player one got it right? " + p1isRight);
         System.out.println("Player two got it right? " + p2isRight);
         System.out.println("Player three got it right? " + p3isRight);
         System.out.println("Game is over.");
         break; // game o   // a játéknak vége, ezért kilépünk a ciklusból

       } else {
         // we mus   // folytatni kell, mert senki nem tippelt helyesen
         System.out.println("Players will have to try again.");
       } // end    // az if/else vége
     } // end     // a ciklus vége
   } // end me    // a tagfüggvény vége
} // end cla
  // az osztály vége
```

A GuessGame három példányváltozóval rendelkezik a három Player objektum számára

létrehozunk három Player objektumot, és hozzárendeljük őket a három Player példányváltozóhoz

bevezetünk három változót a játékosok tippjeinek tárolására

bevezetünk három változót, amelyekben true vagy false értéket tárolunk a játékos tippjétől függően

létrehozzuk a „célszámot", amelyet a játékosoknak ki kell találniuk

meghívjuk a játékosok guess() tagfüggvényét

kiolvassuk a játékosok tippjét (a guess() tagfüggvényük által eredményezett számot) a játékosobjektumok number (szám) változójából

egyenként megvizsgáljuk a játékosok tippjeit, hogy egyezik-e valamelyik a célszámmal. Ha valamelyikük eltalálta a számot, az adott játékos változóját true-ra állítjuk (emlékezz vissza, hogy alapértelmezésként false értéket adtunk nekik)

ha az első VAGY a második VAGY a harmadik játékos jól tippelt... (a || műveleti jel jelentése: VAGY)

ha nincs találat, a ciklusban maradunk, és új tippre kérjük a játékosokat

A kitalálós játék futtatása

```
public class Player {
   int number = 0;  // ide kerül a tipp

   public void guess() {
     number = (int) (Math.random() * 10);
     System.out.println("I'm guessing "
                        + number);
   }
}


public class GameLauncher {
   public static void main (String[] args) {
      GuessGame game = new GuessGame();
      game.startGame();
   }
```

Kimenet (minden futtatáskor különböző lesz)

```
File Edit Window Help Explode
%java GameLauncher
I'm thinking of a number between 0 and 9...
Number to guess is 7
I'm guessing 1
I'm guessing 9
I'm guessing 9
Player one guessed 1
Player two guessed 9
Player three guessed 9
Players will have to try again.
Number to guess is 7
I'm guessing 3
I'm guessing 0
I'm guessing 9
Player one guessed 3
Player two guessed 0
Player three guessed 9
Players will have to try again.
Number to guess is 7
I'm guessing 7
I'm guessing 5
I'm guessing 0
Player one guessed 7
Player two guessed 5
Player three guessed 0
We have a winner!
Player one got it right? true
Player two got it right? false
Player three got it right? false
Game is over.
```



A Java kiviszi a szemetet

A Javában minden alkalommal, amikor létrejön egy objektum, a memória **Kupacnak** nevezett területére kerül. Minden objektum – függetlenül atttól, hogy mikor, hol és hogyan jött létre – a kupacban lakik. Ez azonban nem egy egyszerű memóriakupac: a Java-kupac **Szemétgyűjtő Kupac**. Amikor létrehozol egy objektumot, a Java annak megfelelően foglal memóriaterületet a számára a kupacban, hogy az objektumnak mekkora helyre van szüksége. Egy mondjuk 15 példányváltozóval rendelkező objektumnak valószínűleg több helyre lesz szüksége, mint egy olyannak, amelynek csak 2 példányváltozója van. De mi történik, ha vissza kell szerezned ezt a területet? Hogyan távolíthatsz el egy objektumot a kupacból, ha már nincs rá szükséged? Nos, a Java kezeli helyetted a memóriát. Amikor a JVM „látja", hogy egy objektum használatára már biztosan nem fog többé sor kerülni, *kijelöli azt szemétgyűjtésre*. Ha kevés a memória, a Szemétgyűjtő elindul, kidobja az elérhetetlen objektumokat, és felszabadítja a területet, hogy újrahasznosíthasd azt. A szemétgyűjtő működéséről a későbbi fejezetekből többet is megtudhatsz.

Nincsenek

hülye kérdések

K: Mit tehetek, ha globális változókra és tagfüggvényekre van szükségem? Hogyan oldhatom ezt meg, ha mindennek egy osztályba kell kerülnie?

V: Az objektumközpontú Java-programokban elvileg nincs olyan, hogy „globális" változó vagy tagfüggvény. A gyakorlatban azonban időnként szükség lehet rá, hogy egy tagfüggvény (vagy állandó) a program bármely része számára hozzáférhető legyen. Gondolj a Frázisautomata `random()` tagfüggvényére: ez egy olyan tagfüggvény, amelynek mindenhonnan meghívhatónak kell lennie. Vagy mi a helyzet egy olyan állandóval, mint a *pi*? A 10. fejezetben megtanulod majd, hogy ha egy tagfüggvényt a `public` (nyilvános) és a `static` (statikus) jelzőkkel látsz el, akkor az hasonlóan fog viselkedni, mint a „globális" tagfüggvények. A nyilvános statikus tagfüggvényeket az alkalmazásod bármelyik kódrésze és bármelyik osztálya elérheti. Ha pedig egy változót a `public`, `static` és `final` (végleges) jelzőkkel vezetsz be, akkor lényegében egy globálisan elérhető *állandót* határozol meg.

K: De ha létrehozhatók globális függvények és globális adatok, akkor miért beszélünk objektumközpontúságról?

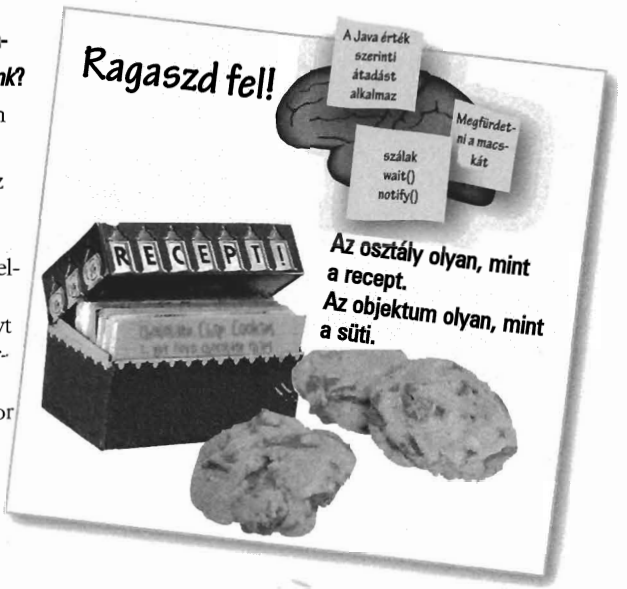
V: Először is, a Javában minden osztályokba kerül. A *pi* állandó és a `random()` tagfüggvény meghatározását – bár nyilvánosak és statikusak – a `Math` osztály tartalmazza. Ezenkívül nem szabad elfelejtened, hogy ezek a statikus („globálisszerű") dolgok inkább kivételt, mintsem szabályt jelentenek a Javában. Különleges esetnek számítanak, amikor is nincs több példány/objektum.

K: Tulajdonképpen *mi* számít Java-programnak? Mi az, amit *terjesztünk*?

V: A Java-programok egy halom (de legalább *egy*) osztályból állnak. A Java-alkalmazásokban az osztályok *egyikének* feltétlenül rendelkeznie kell egy main tagfüggvénnyel, amely a program elindítására szolgál. Programozóként tehát egy vagy több osztályt írsz, és ezeket az osztályokat terjeszted. Ha a végfelhasználó nem rendelkezik JVM-mel, akkor azt is mellékelned kell az alkalmazás osztályaihoz, hogy a felhasználó képes legyen futtatni a programodat. Különféle telepítőprogramok léteznek, amelyeknek a segítségével az osztályaidat többféle (mondjuk különböző rendszerekhez készített) JVM-mel is összecsomagolhatod, és az egészet kiírhatod egy CD-ROM-ra. A végfelhasználónak ezután csak telepítenie kell a JVM megfelelő változatát (feltéve, hogy még nincs Java-értelmezője a számítógépén).

K: Mi van akkor, ha száz vagy ezer osztályom van? Nem körülményes nagyon ennyi önálló fájlt terjeszteni? Nem lehet őket összecsomagolni egyetlen *Alkalmazásban*?

V: Ilyen irdatlan mennyiségű fájlt külön-külön valóban körülményes lenne a végfelhasználókhoz eljuttatni, de erre nincs szükség. Az alkalmazásod valameny-nyi fájlját beleteheted egy Java-archívumba – *egy .jar fájlba* – ami a pkzip formátumot követi. A jar fájlba aztán belefoglalhatsz egy egyszerű, úgynevezett *manifest* formátumú szövegfájlt, amely leírja, hogy a csomag melyik osztálya tartalmazza a futtatandó main() tagfüggvényt.



Újratöltve

- Az objektumközpontú programozás lehetővé teszi, hogy anélkül bővíts egy programot, hogy hozzá kellene nyúlnod a korábban már tesztelt, működő kódokhoz.
- Minden Java-kódot **osztályokban** kell meghatározni.
- Az osztályok azt írják le, hogyan kell létrehozni az adott osztálytípusba tartozó objektumokat. Az **osztály olyan, mint egy tervrajz.**
- Az objektumok gondoskodnak önmagukról – a *hogyan*-ról nem kell tudnod, és nem is kell, hogy érdekeljen.
- Minden objektum **tud** és **csinál** dolgokat
- Azokat a dolgokat, amelyeket egy objektum önmagáról tud, **példányváltozóknak** hívják. Ezek jelképezik az objektum *állapotát* (state).
- Azoknak a dolgoknak, amelyeket egy objektum csinál, **tagfüggvény** a neve. A tagfüggvények az objektum *viselkedését* (behavior) írják le.
- Amikor létrehozol egy osztályt, célszerű egy külön tesztosztályt is írni, amelyet az új osztálytípusba tartozó objektumok létrehozására használhatsz.
- Az osztályok **örökölhetik** egy elvontabb **alaposztály** példányváltozóit és tagfüggvényeit.
- Futáskor a Java-programok csupán egymással társalgó objektumokból állnak.



Gyakorlat

Légy a fordító!



Az ezen az oldalon található Java-fájlok mindegyike egy-egy teljes forrásfájl. A te feladatod az, hogy eljátszd a fordítóprogram szerepét, és kiderítsd, hogy lefordíthatók-e ezek a fájlok. Ha nem, hogyan javítanád ki őket? Ha igen, milyen kimenetet eredményeznek?

A

```
class TapeDeck {

  boolean canRecord = false;

  void playTape() {
    System.out.println("tape playing");
  }

  void recordTape() {
    System.out.println("tape recording");
  }
}

class TapeDeckTestDrive {
  public static void main(String [] args) {

    t.canRecord = true;
    t.playTape();

    if (t.canRecord == true) {
      t.recordTape();
    }
  }
```

B

```
class DVDPlayer {

  boolean canRecord = false;

  void recordDVD() {
    System.out.println("DVD recording");
  }
}

class DVDPlayerTestDrive {
  public static void main(String [] args) {

    DVDPlayer d = new DVDPlayer();
    d.canRecord = true;
    d.playDVD();

    if (d.canRecord == true) {
      d.recordDVD();
    }
  }
}
```



Gyakorlat

Kódmágnesek



Egy Java-program kódja összekeveredett a hűtőmágneseken. Össze tudod rakni úgy a kódrészleteket, hogy előálljon belőle a lap alján látható kimenetet eredményező Java-program? Néhány kapcsos zárójel leesett a padlóra, és túl kicsi volt ahhoz, hogy fel tudjuk venni, úgyhogy nyugodtan egészítsd ki a kódot annyi kapcsos zárójellel, amennyit csak szükségesnek érzel!

```
d.playSnare();
```

```
DrumKit d = new DrumKit();
```

```
boolean topHat = true;
boolean snare = true;
```

```
void playSnare() {
  System.out.println("bang bang ba-bang");
}
```

```
public static void main(String [] args) {
```

```
if (d.snare == true) {
  d.playSnare();
}
```

```
d.snare = false;
```

```
class DrumKitTestDrive {
```

```
d.playTopHat();
```

```
class DrumKit {
```

```
void playTopHat () {
  System.out.println("ding ding da-ding");
}
```

```
File Edit Window Help Dance
% java DrumKitTestDrive
bang bang ba-bang
ding ding da-ding
```



Halászd ki a medencéből!



A ***feladatod*** az, hogy kihalászd a kóddarabokat a medencéből, és elhelyezd azokat a kód üresen hagyott helyein. Egy kóddarabot egynél **többször** is felhasználhatsz, viszont nem lesz szükséged minden darabra. A ***cél*** olyan osztályok felépítése, amelyek lefordíthatók, futtathatók, és az alább látható kimenetet eredményezik.

Kimenet

```
File  Edit  Window  Help  Implode
%java EchoTestDrive
helloooo...
helloooo...
helloooo...
helloooo...
10
```

```
public class EchoTestDrive {
  public static void main(String [] args) {
    Echo e1 = new Echo();

    _____________________

    int x = 0;
    while ( _____________ ) {
      e1.hello();

      ___________________________

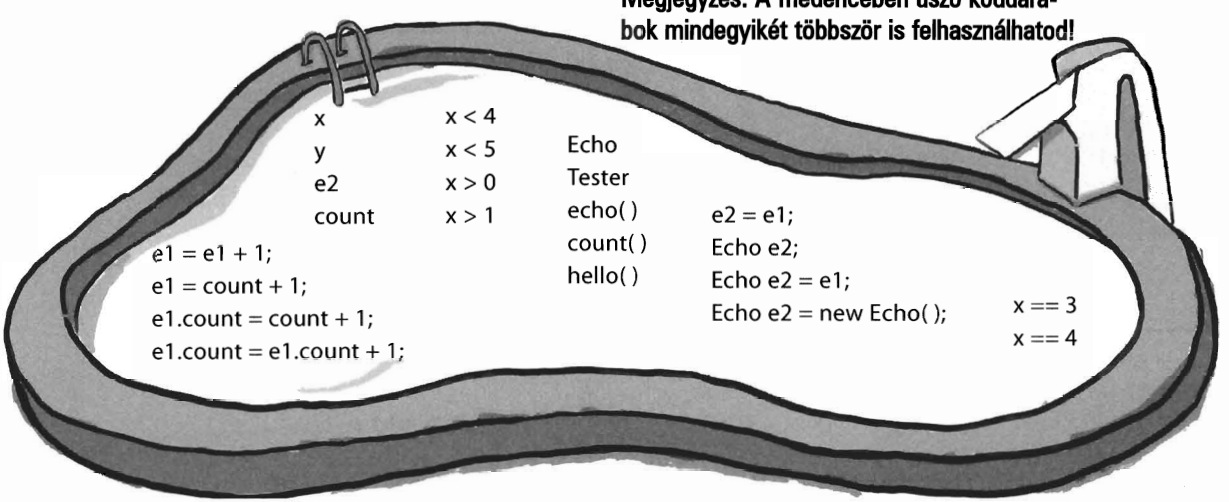
      if ( _____________ ) {
        e2.count = e2.count + 1;
      }
      if ( _____________ ) {
        e2.count = e2.count + e1.count;
      }
      x = x + 1;
    }
    System.out.println(e2.count);
  }
}
```

```
class _____________ {
  int _____________ = 0;
  void _____________ {
    System.out.println("helloooo... ");
  }
}
```

Bónusz kérdés!

Ha a kimenet utolsó sora **10** helyett **24** lenne, hogyan raknád össze a darabokat?

Megjegyzés: A medencében úszó kóddarabok mindegyikét többször is felhasználhatod!





Álarcosbál



Néhány Java-összetevő kosztümbe öltözve álarcosbált tart. Adnak neked egy kis segítséget, és ki kell találnod, hogy ki kicsoda. Tegyük fel, hogy mindig igazat állítanak magukról. Ha olyasmit találnak mondani, ami többükre is jellemző lehet, válaszd ki mindegyiket, akire ráillik a leírás, és írd a nevüket az adott állítás mellé. Az első vendéget leleplezzük neked.

A mai vendégek:

Osztály **Tagfüggvény** **Objektum** **Példányváltozó**

Engem egy .java fájlból hoz létre a fordító. *osztály*

A példányváltozóim értéke különbözhet a barátaimétól. _____

Úgy viselkedem, mint egy sablon. _____

Csinálni szeretek dolgokat. _____

Nekem több tagfüggvényem is lehet. _____

Én az „állapotot" ábrázolom. _____

Nekem viselkedésem van. _____

Objektumokban találsz meg. _____

A kupacban lakom. _____

Engem objektumpéldányok létrehozására használnak. _____

Az állapotom megváltozhat. _____

Tagfüggvényeket vezetek be. _____

Futás közben megváltozhatok. _____



Gyakorlat – Megoldás

Kódmágnesek:

```
class DrumKit {

  boolean topHat = true;
  boolean snare = true;

  void playTopHat() {
    System.out.println("ding ding da-ding");
  }

  void playSnare() {
    System.out.println("bang bang ba-bang");
  }
}

class DrumKitTestDrive {
  public static void main(String [] args) {

    DrumKit d = new DrumKit();
    d.playSnare();
    d.snare = false;
    d.playTopHat();

    if (d.snare == true) {
      d.playSnare();
    }
  }
}
```

```
File Edit Window Help Dance
% java DrumKitTestDrive
bang bang ba-bang
ding ding da-ding
```

Légy a fordító!

A

```
class TapeDeck {
  boolean canRecord = false;
  void playTape() {
    System.out.println("tape playing");
  }
  void recordTape() {
    System.out.println("tape recording");
  }
}

class TapeDeckTestDrive {
  public static void main(String [] args) {

    TapeDeck t = new TapeDeck( );
    t.canRecord = true;
    t.playTape();

    if (t.canRecord == true) {
      t.recordTape();
    }
  }
}
```

Megvan a sablon, most létre kell hoznunk egy objektumot!

B

```
class DVDPlayer {
  boolean canRecord = false;
  void recordDVD() {
    System.out.println("DVD recording");
  }
  void playDVD ( ) {
    System.out.println("DVD playing");
  }
}

class DVDPlayerTestDrive {
  public static void main(String [] args) {
    DVDPlayer d = new DVDPlayer();
    d.canRecord = true;
    d.playDVD();
    if (d.canRecord == true) {
      d.recordDVD();
    }
  }
}
```

A d.playDVD(); sor tagfüggvény nélkül nem fordítható le!



Fejtörők – Megoldás

Halászd ki a medencéből!

```
public class EchoTestDrive {
  public static void main(String [] args) {
    Echo e1 = new Echo();
    Echo e2 = new Echo(); // a helyes válasz
        – vagy –
    Echo e2 = e1; // válasz a bónusz kérdésre
    int x = 0;
    while ( x < 4 ) {
      e1.hello();
      e1.count = e1.count + 1;
      if ( x == 3 ) {
        e2.count = e2.count + 1;
      }
      if ( x > 0 ) {
        e2.count = e2.count + e1.count;
      }
      x = x + 1;
    }
    System.out.println(e2.count);
  }
}
```

```
class Echo {
  int count = 0;
  void hello( ) {
    System.out.println("helloooo... ");
  }
}
```

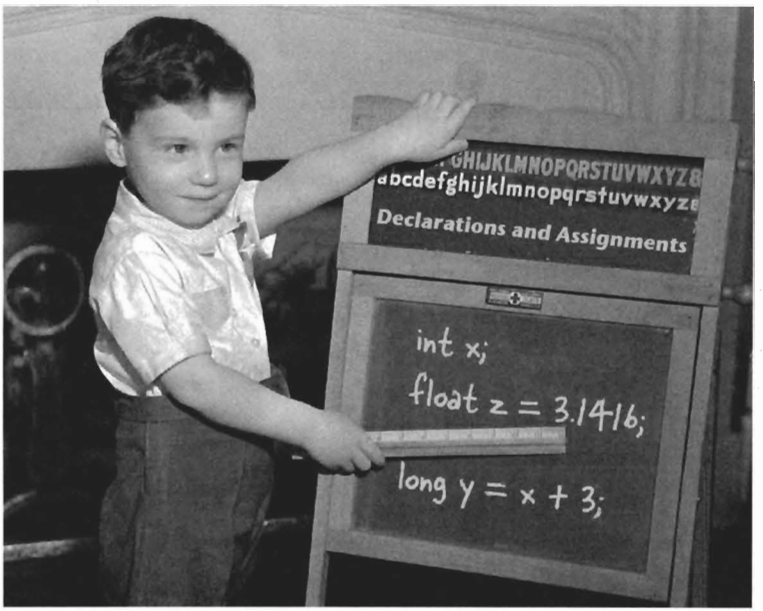
```
File  Edit  Window  Help  Assimilate
%java EchoTestDrive
helloooo...
helloooo...
helloooo...
helloooo...
10
```

Álarcosbál

Engem egy .java fájlból hoz létre a fordító.	*osztály*
A példányváltozóim értéke különbözhet a barátaimétól.	*objektum*
Úgy viselkedem, mint egy sablon.	*osztály*
Csinálni szeretek dolgokat.	*objektum, tagfüggvény*
Nekem több tagfüggvényem is lehet.	*osztály, objektum*
Én az „állapotot" ábrázolom.	*példányváltozó*
Nekem viselkedésem van.	*objektum, osztály*
Objektumokban találsz meg.	*tagfüggvény, példányváltozó*
A kupacban lakom.	*objektum*
Engem objektumpéldányok létrehozására használnak.	*osztály*
Az állapotom megváltozhat.	*objektum, példányváltozó*
Tagfüggvényeket vezetek be.	*osztály*
Futás közben megváltozhatok.	*objektum, példányváltozó*

Megjegyzés: az osztályoknak és az objektumoknak egyaránt van állapota és viselkedése. Az osztályban „meghatározzuk" (define) őket, az objektum pedig „rendelkezik velük" (have). Az, hogy *valójában* hol laknak, most nem érdekes a számunkra.

Ismerd meg a változóidat!



A változóknak két fajtája létezik: a primitívek (alaptípusok) és a hivatkozások. Eddig két helyen használtál változókat: az objektum **állapotának** leírására (példányváltozók) és **helyi változókként** (*tagfüggvényen* belül bevezetett változók). Később majd **argumentumokként** (a hívó kód által a meghívott tagfüggvénynek átadott értékekként) és **visszaadott típusként** (a tagfüggvény hívójának visszaküldött értékekként) is alkalmazni fogod őket. Láttál már egyszerű alaptípusba tartozóként (**primitívként**) bevezetett egész értékű (`int` típusú) változókat, és összetettebb változókat, például karakterláncokat és tömböket is. Az **élet azonban többről kell, hogy szóljon** az egészeknél, karakterláncoknál és tömböknél. Mi van akkor, ha van egy PetOwner (Gazdi) objektumod egy Dog (Kutya) példányváltozóval? Vagy egy Car (Autó) objektumod egy Engine-nel (Motor)? Ebben a fejezetben lerántjuk a leplet a Java típusairól, és megnézzük, mi az, amit változóként *bevezethetsz*, mi az, amit egy változóba *beletehetsz*, és mi az, amit a változókkal *csinálhatsz*. Ezenkívül végre látni fogod, milyen az élet *valójában* a szemétgyűjtő kupacban.

Változó bevezetése



A Java odafigyel a típusokra. Nem engedi, hogy olyan bizarr és veszélyes dolgokat művelj, mint hogy egy Rabbit (Nyúl) változóba egy Giraffe (Zsiráf) hivatkozást gyömöszölsz – mi történik például, ha valaki arra próbálja utasítani a „zsiráfot", hogy ugorjon (`hop()`)? A Java azt sem hagyja, hogy egy egész típusú változóba lebegőpontos számot tegyél, hacsak *be nem vallod a fordítónak*, hogy tisztában vagy vele, hogy így pontosságot veszíthetsz (mindent, ami a tizedesvessző után áll).

A fordítóprogram a legtöbb problémát képes kiszúrni:

```
Rabbit hopper = new Giraffe();
```

Ne várd, hogy ezt lefordítsa. *Szerencsére* nem sikerülhet. Ahhoz, hogy a típusbiztonság érvényesüljön, minden változót a típusával együtt kell bevezetned. Egész változóról van szó? Egy Kutyáról? Egyetlen karakterről? A változóknak két fajtája létezik: ***primitívek*** (***alaptípusok***) és ***objektumhivatkozások***. A primitívek alaptípusú értékeket (vagyis egyszerű bitmintákat) tárolnak – egészeket, logikai értékeket, lebegőpontos számokat –, míg az objektumhivatkozások (minő meglepetés) *objektumokra* mutató *hivatkozásokat* (nahát!).

Először a primitíveket vizsgáljuk meg, majd tisztázzuk, mit is jelent az, hogy objektumhivatkozás. A változó fajtájától függetlenül azonban két szabályt mindig be kell tartanod, amikor bevezetsz egy változót:

Minden változónak meg kell adnod a típusát.

A típus mellett a változóknak névre is szükségük van, mert a kódban a nevükkel hivatkozhatsz rájuk.

Minden változónak adnod kell egy nevet.

```
int count;
```

típus — név

Megjegyzés: Ha egy olyan utasítást látsz, aminek a jelentése „X **típusú** objektum", a *típust* az *osztály* szinonimájának tekintheted. (A későbbi fejezetekben ezen az állításon finomítunk egy kicsit.)

„Egy dupla tejeskávét kérek – nem is, legyen inkább egy int”

A Java-változókra gondolhatsz úgy, mintha csészék, bögrék vagy korsók lennének: kávéscsészék; teásbögrék; söröskorsók sok-sok habzó nedűvel; pattogatott kukoricával töltött vödrök, amilyeneket a moziban adnak; csészék gömbölyű, szexi füllel; vagy fémperemű csészék, amelyekről tudod, hogy soha, de soha nem szabad betenni őket a mikróba.

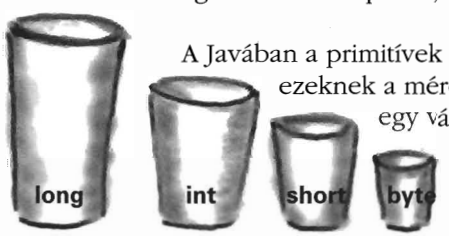
A változó csupán egy csésze. Egy tároló, ami *tárol* valamit.

A csészékhez hasonlóan a változóknak is van mérete és típusa. Ebben a fejezetben először azokat a változókat (csészéket) tekintjük át, amelyek **primitíveket** tárolnak, egy kicsit később pedig azokat, amelyek *objektumokra mutató hivatkozásokat*. Végig a csészehasonlatnál maradunk – bármilyen bárgyúnak tűnik is ebben a pillanatban a párhuzam, sokat segít majd, amikor jobban belebonyolódunk a tárgyalásba (ami hamarosan bekövetkezik).

A primitívek olyanok, mint a kávézókban található csészék. Ha jártál már Starbucks kávézóban, akkor tudod, miről beszélünk: azokról a különféle méretű csészékről, amelyek mindegyikéhez társul egy „név”: „rövid”, „hosszú”, „Szeretnék egy 'grande' mochát, csökkentett koffeintartalmút, extra tejszínhabbal”.

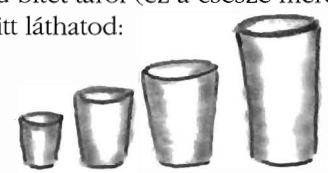


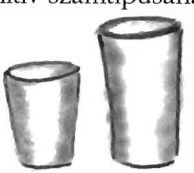
A csészéket esetleg ki is teszik a pultra, hogy segítsenek a rendelésben:



A Javában a primitívek (alaptípusok) különböző méretűek, és ezeknek a méreteknek is van neve. Amikor bevezetsz egy változót a Javában, meg kell adnod a konkrét típusát. Az itt látható négy tároló a Java négy egész alaptípusát ábrázolja:

Minden csésze egy értéket tárol, csak éppen a Java-primitívek esetében nem azt mondod a fordítónak, hogy „Egy francia pörkölésű hosszút kérek”, hanem azt, hogy „Egy int változót legyen szíves, benne a 90-es számmal”. És van még egy aprócska különbség: a Javában *nevet* is kell adnod a csészéknek. A teljes rendelés tehát így hangzik: „Egy int változót legyen szíves, benne a 2486-os számmal, és a változó neve legyen ***height***”. Minden primitív változó rögzített számú bitet tárol (ez a csésze mérete). A Java hat primitív számtípusának a méretét itt láthatod:





Alaptípusok

Típus	Bitmélység	Értéktartomány
boolean és char		
boolean	(JVM-függő)	*true* vagy *false*
char	16 bit	0-tól 65535-ig
számtípusok (mindegyik előjeles)		
egész		
byte	8 bit	-128-tól 127-ig
short	16 bit	-32768-tól 32767-ig
int	32 bit	-2147483648-tól 2147483647-ig
long	64 bit	-hatalmastól +hatalmasig
lebegőpontos		
float	32 bit	változó
double	64 bit	változó

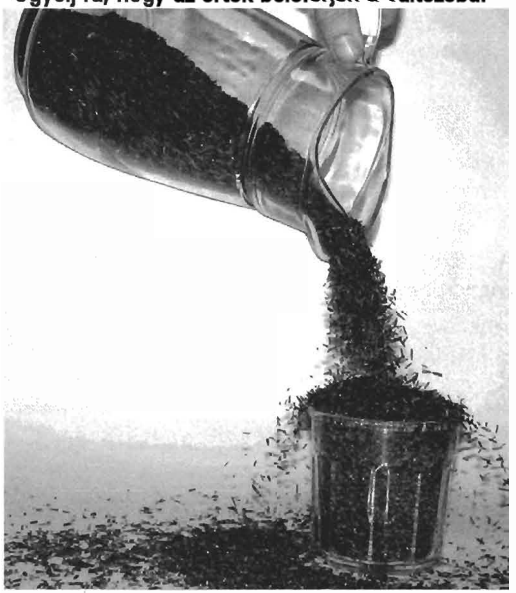
Primitívek bevezetése értékadással:

```
int x;
x = 234;
byte b = 89;
boolean isFun = true;
double d = 3456.98;
char c='f';
int z = x;
boolean isPunkRock;
isPunkRock = false;
boolean powerOn;
powerOn = isFun;
long big = 3456789;
float f = 32.5f;
```

Figyeld meg az 'f'-et. Ezt a float értékeknél ki kell tenned, mert a Java minden lebegőpontos értéket double típusúként kezel, hacsak nem 'f'-ként adod meg

Nem szeretnéd, ha kicsordulna...

Ügyelj rá, hogy az érték beleférjen a változóba.



Egy kis csészébe nem tehetsz nagy értéket.
Na jó, tehetsz, de akkor *túlcsordul* a tetején. A fordítóprogram ezt megpróbálja megelőzni, ha látja a kódodból, hogy valami nem fog beleférni a kért tárolóba (változóba/csészébe).
Egy int-nyi dolgot nem tölthetsz például egy byte méretű tárolóba:

```
int x = 24;
byte b = x;
  // nem fog sikerülni!!
```

Miért nem sikerül? Végtére is, az *x* értéke 24, a 24 pedig határozottan elég kicsi ahhoz, hogy beleférjen egy byte-ba. *Te* is tudod, *mi* is tudjuk, de a fordítót csak az érdekli, hogy valami nagyot próbálsz beletenni egy kicsi tárolóba, és ezért fennáll a *lehetősége* a túlcsordulásnak. Nem várhatod el a fordítótól, hogy ismerje az *x* értékét, még ha konkrétan látod is az értéket a kódban.

Egy változóhoz többféleképpen is hozzárendelhetsz egy értéket:

- beírsz egy konkrét („literális") értéket az egyenlőségjel után (x= ***12***, isGood = ***true*** stb.)
- egy változó értékét rendeled hozzá egy másik változóhoz (x = y)
- egy kifejezés segítségével kombinálod a kettőt (x = y + ***43***)

Az alábbi példákban a literális értékeket félkövér, dőlt betűkkel jelöltük:

`int size = 32;` egy *size* nevű, int típusú változót vezet be, és a *32* értéket rendeli hozzá

`char initial = 'j' ;` egy *initial* (kezdőbetű) nevű, char típusú változót vezet be, és a *'j'* értéket rendeli hozzá

`double d = 456.709;` egy *d* nevű, double típusú változót vezet be, a *456,709* értékkel

`boolean isCrazy;` egy *isCrazy* nevű, boolean típusú változót vezet be (értékadás nélkül)

`isCrazy = true;` a *true* értéket rendeli a korábban bevezetett *isCrazy* változóhoz

`int y = x + 456;` egy *y* nevű, int típusú változót vezet be, és az *x* jelenlegi értékének, valamint a *456* értéknek az összeadásából származó értéket rendeli hozzá

Hegyezd ki a ceruzád!

A fordítóprogram nem engedi, hogy egy nagy csészében levő értéket egy kis csészébe önts. De mi a helyzet, ha fordítva próbálod – egy kicsit töltesz át egy nagyba? ***Semmi gond.*** Annak alapján, amit a primitív változók méretéről és típusáról megtanultál, próbáld kitalálni, hogy az alábbi értékadások közül melyik érvényes, és melyik nem az. Még nem ejtettünk szót minden szabályról, ezért itt-ott a józan eszedre kell támaszkodnod. ***Tipp:*** a fordító mindig a biztonságot részesíti előnyben.

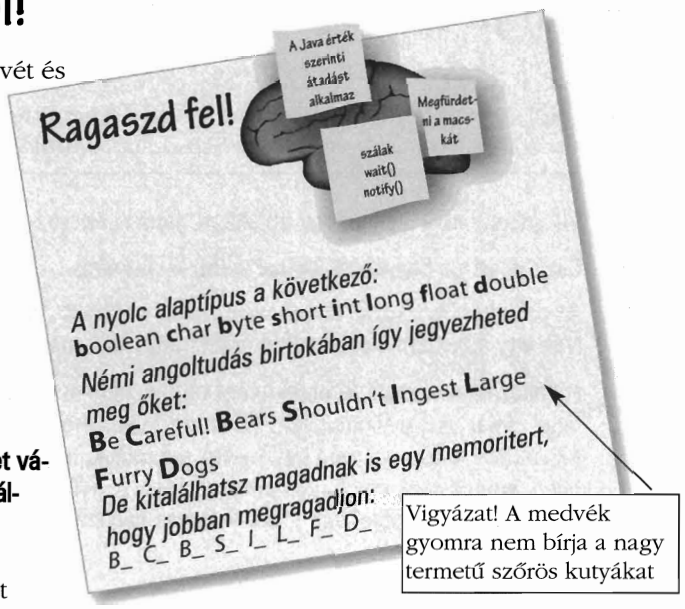
Az alábbi felsorolásban karikázd be azokat az utasításokat, amelyek egy adott tagfüggvényen belül érvényesek lennének:

```
1. int x = 34.5;
2. boolean boo = x;
3. int g = 17;
4. int y = g;
5. y = y + 10;
6. short s;
7. s = y;
8. byte b = 3;
9. byte v = b;
10. short n = 12;
11. v = n;
12. byte k = 128;
```

El a kezekkel attól a kulcsszótól!

Azt már tudod, hogy a változóidnak meg kell adnod a nevét és a típusát. Már ismered az alaptípusokat (primitíveket).
De mit használhatsz névként? Nos, a szabályok egyszerűek. Egy osztályt, tagfüggvényt vagy változót az alábbi szabályok szerint nevezhetsz el (a szabályok valójában egy kicsit rugalmasabbak, de így biztonságosabb):

- **A névnek betűvel, aláhúzásjellel (_) vagy dollárjellel ($) kell kezdődnie. A nevek nem kezdődhetnek számmal.**
- **Az első karakter után számokat is írhatsz; azok csak a név elején tiltottak.**
- **Feltéve, hogy betartod a fenti két szabályt, tetszőleges nevet választhatsz, de a Java foglalt (fenntartott) szavait nem használhatod.**



Vannak úgynevezett kulcsszavak (és egyebek), amelyeket a fordítóprogram felismer. Ha össze szeretnéd zavarni a fordítót (nem jó ötlet), *próbáld ki*, hogy névként egy foglalt szót adsz meg.
Már találkoztál néhány foglalt szóval, amikor megírtad az első osztályodat:

```
public   static   void
```

A primitív típusok nevei ugyancsak a fenntartott szavak közé tartoznak:

```
boolean char byte short int long float double
```

a saját változóneveidben ezek közül egyiket se használd



És még sok foglalt szó van, amivel eddig nem találkoztál. Ha nem is kell tudnod, mit jelentenek, azzal tisztában kell lenned, hogy te nem használhatod őket. *Semmilyen körülmények között* ***ne próbáld most megjegyezni ezeket a szavakat***, mert ahhoz, hogy helyet csinálj nekik a fejedben, egy csomó mindent ki kell dobálnod (például hogy hová parkoltál a kocsiddal). Ne aggódj, mire a könyv végére érsz, a legtöbbet megtanulod majd.

Ez a táblázat foglalt.

boolean	byte	char	double	float	int	long	short	public	private
protected	abstract	final	native	static	strictfp	synchronized	transient	volatile	if
else	do	while	switch	case	default	for	break	continue	assert
class	extends	implements	import	instanceof	interface	new	package	super	this
catch	finally	try	throw	throws	return	void	const	goto	enum

A Java kulcsszavai és foglalt szavai (fontossági sorrend nélkül). Ha ezeket a szavakat használod, a fordítóprogram nagyon, *nagyon* dühös lesz.

A Kutya objektumod vezérlése

Már tudod, hogyan kell bevezetni egy primitív változót, és hogyan kell értéket adni neki. De mi a helyzet a nem primitív típusú változókkal? Más szavakkal, *mi van az objektumokkal?*

- **Valójában nincs olyan, hogy objektum típusú változó.**
- **Csak objektumhivatkozás típusú változók léteznek.**
- **Az objektumhivatkozás típusú változók olyan biteket tárolnak, amelyek egy objektum elérésének módját határozzák meg.**
- **A változó nem magát az objektumot tárolja, hanem egy mutatót vagy címet. A Javában ugyanakkor nem tudjuk, hogy *mi* van egy hivatkozás típusú változón belül. Csak azt tudjuk, hogy bármi is legyen, egy és csak egy objektumot jelképez. A JVM viszont tudja, hogyan érje el az objektumot a hivatkozás segítségével.**

Egy objektumot nem gyömöszölhetsz bele egy változóba. Gyakran így gondolunk rá, és olyasmiket mondunk, hogy „Átadtam a String-et a System.out.println() tagfüggvénynek", vagy „A tagfüggvény egy Kutyát ad vissza", vagy „Egy új Foo objektumot tettem a myFoo nevű változóba" – de valójában nem ez történik.

Nincsenek hatalmas bővíthető csészék, amelyek tetszőleges objektum méretére képesek nőni. Az objektumok egyetlen helyen laknak – a szemétgyűjtő kupacban. (Erről bővebben beszélünk a fejezet későbbi részében.)

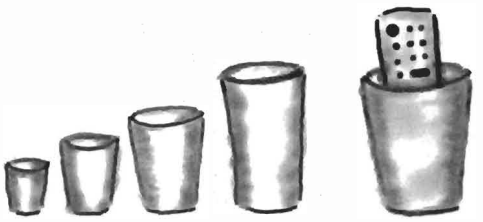
A primitív változók olyan bitekkel vannak tele, amelyek a változó tényleges ***értékét*** ábrázolják, az objektumhivatkozás típusú változók azonban ***az objektum elérésének módját*** tárolják. A hivatkozó változók esetében a pont műveletet (.) használhatod arra, hogy azt mondd, „fogd a pont *előtti* dolgot, hogy megszerezd nekem a pont *utáni*t":

```
myDog.bark();
```

Ez azt jelenti, hogy „fogd a myDog változó által hivatkozott objektumot, és hívd meg a segítségével a bark() tagfüggvényt". Amikor a pont műveletet használod egy objektumhivatkozáson, az olyan, mintha megnyomnál egy gombot az objektum távirányítóján.



A Dog (Kutya) hivatkozó változót tekintsd a Kutya távirányítójának, amivel ráveheted az „objektumot", hogy különböző dolgokat csináljon (tagfüggvényeket hívjon meg).



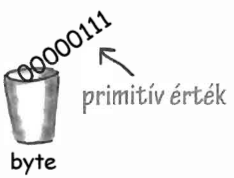
Az objektumhivatkozások is csak változóértékek.

Olyasmi, amit beletölthetsz egy csészébe – csak éppen az érték ezúttal egy távirányító.

Primitív típusú változó

```
byte x = 7;
```

A változóba a 7-et ábrázoló bitek (00000111) kerülnek.



Hivatkozó változó

```
Dog myDog = new Dog();
```

A változóba a Dog objektum elérésének módját ábrázoló bitek kerülnek.

Maga a Dog objektum nem kerül a változóba!



A primitív változók esetében a változó értéke... az *érték* (5, -26.7, 'a').
A hivatkozó változók esetében a változó értéke... *az adott objektum elérésének módját ábrázoló bitek.*
Nem kell tudnod, hogyan valósítja meg a JVM az objektumhivatkozásokat. Persze egy mutatóval, ami egy mutatóra mutat stb. – de még ha tudod is, a biteket semmi másra nem használhatod, csak egy objektum elérésére.

Azzal nem foglalkozunk, hogy hány 1-es és 0 van egy hivatkozás típusú változóban. Ez a JVM-től és a hold fázisától függ.

Az objektumok bevezetésének, létrehozásának és értékkel való ellátásának 3 lépése

```
1        3  2
Dog myDog = new Dog();
```

1 Hivatkozó változó bevezetése

```
Dog myDog = new Dog();
```

Ez arra utasítja a JVM-et, hogy foglaljon helyet egy hivatkozás típusú változónak, és adja neki a *myDog* nevet. A hivatkozás örökre Dog típusú lesz. Más szavakkal, egy olyan távirányító, amelyen egy kutya (Dog) – és nem egy macska (Cat), gomb (Button) vagy csatoló (Socket) – vezérlésére szolgáló gombok vannak.



2 Objektum létrehozása

```
Dog myDog = new Dog();
```

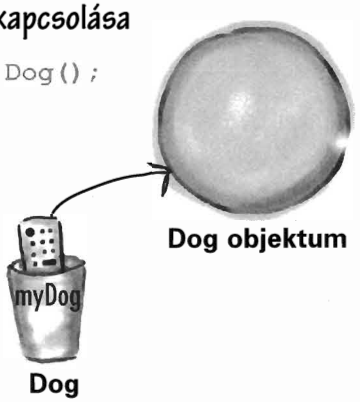
Ez arra utasítja a JVM-et, hogy foglaljon helyet egy új Dog objektum számára a kupacban (erről később bővebben is beszélünk, különösen a 9. fejezetben).



3 Az objektum és a hivatkozás összekapcsolása

```
Dog myDog = new Dog();
```

Ez az utasítás az új Dog-ot a myDog nevű hivatkozó változóhoz rendeli. Más szavakkal, ***beprogramozza a távirányítót***.



Nincsenek
hülye kérdések

K: Milyen nagy egy hivatkozó változó?

V: Azt nem tudhatod. Hacsak nem vagy jóban valakivel a JVM fejlesztőcsapatából, nem ismerheted a hivatkozások ábrázolási módját. Valahol biztos mutatók bújnak meg mögöttük, de nem férhetsz hozzájuk. Nincs is rá szükséged. (Na jó, ha ragaszkodsz hozzá, képzeld azt, hogy 64 bites értékekről van szó.) Memóriafoglalási ügyekben a Nagy Kérdés az, hogy hány *objektumot* (és nem objektum*hivatkozást*) hozol létre, és *azok* (az *objektumok*) milyen nagyok.

K: Ez azt jelenti, hogy minden objektumhivatkozás azonos méretű, függetlenül attól, hogy ténylegesen mekkorák azok az objektumok, amelyekre hivatkoznak?

V: Aha. Egy adott JVM esetében minden hivatkozásnak egyforma a mérete, függetlenül a hivatkozott objektumoktól, de mivel minden JVM másképp ábrázolhatja a hivatkozásokat, a hivatkozások kisebbek vagy nagyobbak lehetnek az egyik JVM-en, mint egy másikon.

K: A hivatkozó változókon is lehet matematikai műveleteket végezni, például növelni őket? Tudjátok, ilyen C-dolgokat...

V: Nem. Ismételd velünk: „A Java nem C."



Java – lehull a lepel

Heti interjúalanyunk: Hivatkozás

Agyhullám: Elmondaná, milyen egy objektumhivatkozás élete?

Hivatkozás: Ó, nincs benne semmi érdekes. Én csak egy távirányító vagyok, amit be lehet programozni, hogy különböző objektumokat vezéreljen.

Agyhullám: A különböző objektumokat úgy érti, hogy miközben Ön fut? Például képes egy Kutyára hivatkozni, aztán öt perccel később egy Autóra?

Hivatkozás: Természetesen nem. Ha bevezettek, akkor már nincs mit tenni. Ha kutya-távirányító vagyok, akkor soha nem leszek képes semmi másra mutatni (húha, elszóltam magam; nem mondhatom, hogy *mutatni*), úgy értem, *hivatkozni*, mint egy Kutyára.

Agyhullám: Ez azt jelenti, hogy csak egy Kutyára hivatkozhat?

Hivatkozás: Nem. Hivatkozhatok egy bizonyos Kutyára, aztán öt perc múlva egy *másik* Kutyára. Amíg Kutyáról van szó, addig át lehet irányítani egy másikra (ahogy a távirányítót is át lehet programozni egy másik TV-re). Hacsak...á, hagyjuk.

Agyhullám: Nem, nem, ez érdekes. Mit akart mondani?

Hivatkozás: Nem hiszem, hogy érdemes ebbe belebonyolódni, de azért elmondom röviden. Ha `final` jelzőt kapok, akkor semmi másra nem lehet beprogramozni, csak *arra* a bizonyos Kutyára. Más szavakkal, semmilyen más objektumot nem lehet hozzám rendelni.

Agyhullám: Igaza van, ebbe most tényleg ne menjünk bele. Szóval, hacsak nem *végleges* (`final`) Ön, akkor hivatkozhat egy Kutyára, aztán később egy másikra. Azt is megteheti, hogy *semmire* nem hivatkozik? Be lehet programozni a „semmire"?

Hivatkozás: Igen, de erről nem szeretnék beszélni.

Agyhullám: Miért?

Hivatkozás: Mert ez azt jelenti, hogy egy `null` vagyok, ami nagyon kínos nekem.

Agyhullám: Úgy érti, azért, mert akkor nincs értéke?

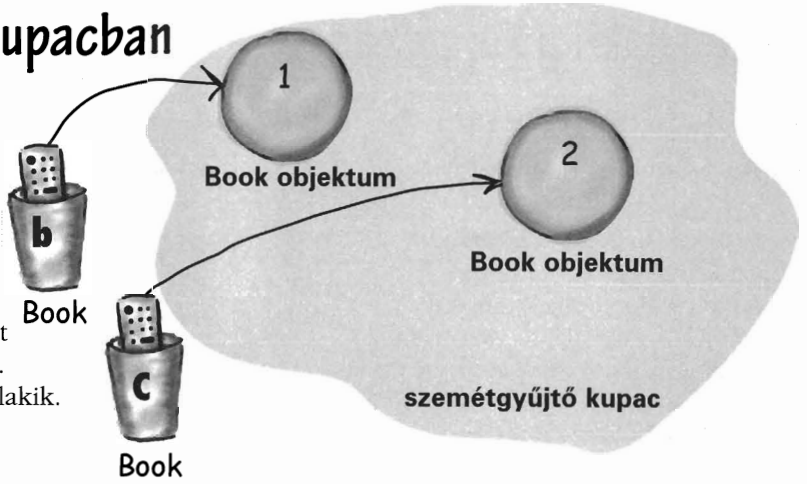
Hivatkozás: Ó, nem. A `null` is egy érték. Attól még távirányító maradok, de olyan, mintha venne egy új, univerzális távirányítót, miközben nincs is tévéje. Nem irányíthatok semmit. Egész nap nyomkodhatják a gombjaimat, semmi érdekes nem történik. Ettől olyan...feleslegesnek érzem magam. Elpazarolt biteknek. Persze nem túl sok bitnek, de akkor is. És ez még nem a legrosszabb. Ha én vagyok az egyetlen hivatkozás egy bizonyos objektumra, és valaki `null`-ra állít („kiprogramoz"), akkor az azt jelenti, hogy többé *senki* nem érheti el azt az objektumot, amelyre hivatkoztam.

Agyhullám: És ez szörnyű, mert...?

Hivatkozás: Még kérdeznie kell? Felépítettem egy kapcsolatot ezzel az objektummal, bizalmas viszonyba kerültem vele, és erre valaki hirtelen elvágja a köztünk levő köteléket. Ráadásul soha többé nem láthatom az objektumot, mert kijelölődik [Megjegyzés a gyártásvezetőnek: tragikus zene bevág] *szemétgyűjtésre.* [Zokogás] Azt hiszi, a programozók valaha is gondolnak *erre?* [Szipogás] Miért? *Miért* nem lehetek primitív? *Utálok hivatkozás lenni!* A felelősség, a tönkretett kapcsolatok...

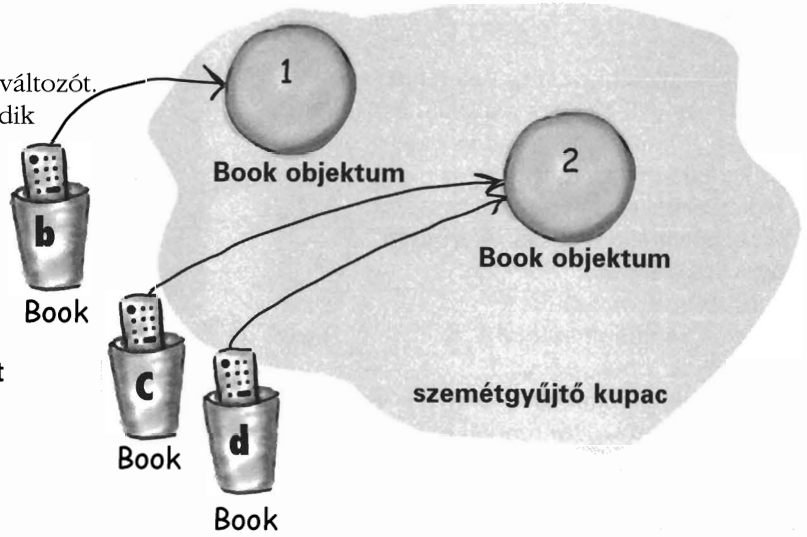
Élet a szemétgyűjtő kupacban

```
Book b = new Book();
Book c = new Book();
```



Bevezetsz két Book (Könyv) nevű hivatkozó változót. Létrehozol két új Book objektumot. A Book objektumokat hozzárendeled a hivatkozó változókhoz. A két Book objektum most a kupacban lakik.
Hivatkozások száma: 2
Objektumok száma: 2

```
Book d = c;
```



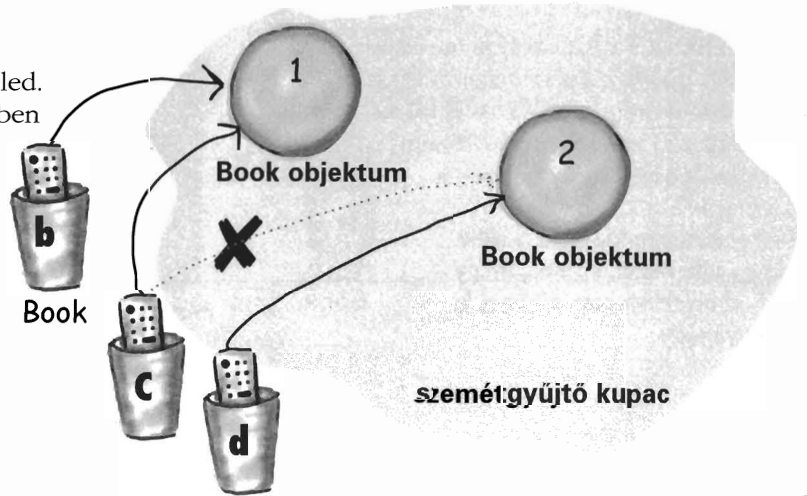
Bevezetsz egy új, Book nevű hivatkozó változót. Ahelyet, hogy létrehoznál egy új, harmadik Book objektumot, a ***c*** változó értékét rendeled a ***d*** változóhoz. Mit jelent ez? Nos, olyan, mintha azt mondanád, hogy „Fogd a biteket ***c***-ből, másold le őket, és tedd a másolatot ***d***-be".

A *c* és a *d* ugyanarra az objektumra hivatkozik.

A *c* és *d* változók ugyanannak az értéknek két különböző példányát tárolják – két távirányító ugyanarra a TV-re beprogramozva.

Hivatkozások száma: 3
Objektumok száma: 2

```
c = b;
```



A ***b*** változó értékét a ***c*** változóhoz rendeled. Most már tudod, hogy ez mit jelent: a ***b***-ben levő bitekről másolat készül, és ez a másolat kerül ***c***-be.

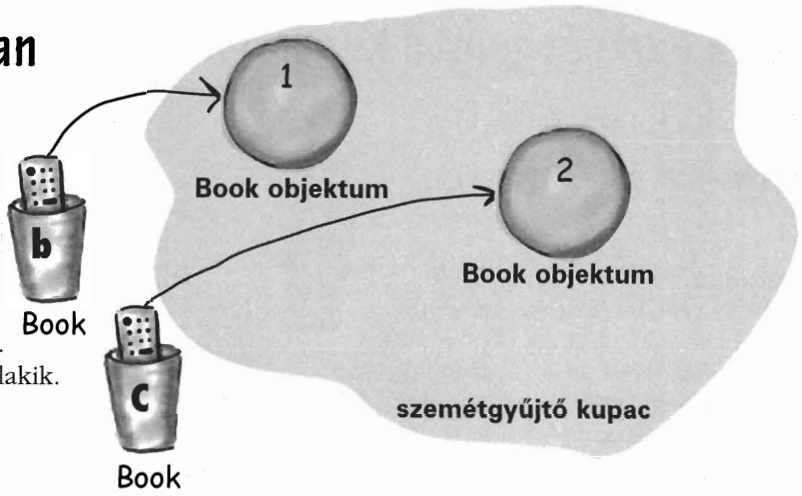
A *b* és a *c* ugyanarra az objektumra hivatkozik.

Hivatkozások száma: 3
Objektumok száma: 2

Élet és halál a kupacban

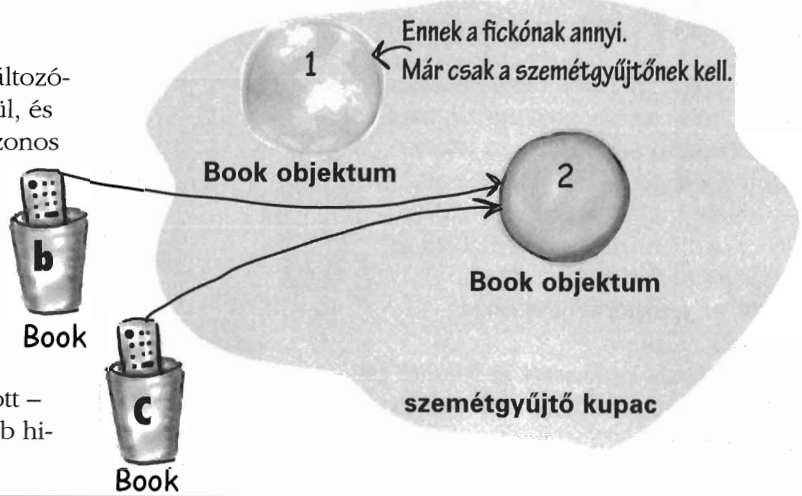
```
Book b = new Book();
Book c = new Book();
```

Bevezetsz két Book nevű hivatkozó változót. Létrehozol két új Book objektumot. A Book objektumokat hozzárendeled a hivatkozó változókhoz. A két Book objektum most a kupacban lakik.
Aktív hivatkozások száma: 2
Elérhető objektumok száma: 2



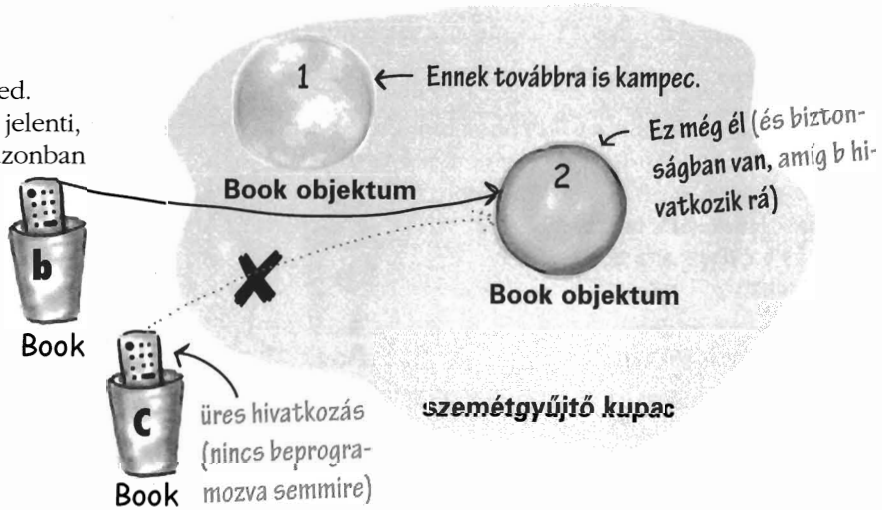
```
b = c;
```

A **c** változó értékét hozzárendeled a **b** változóhoz. A **c**-ben levő bitekről másolat készül, és ez a másolat kerül **b**-be. A két változó azonos értékeket tartalmaz.
A *b* és a *c* ugyanarra az objektumra hivatkozik. Az 1-es objektum kijelölődik szemétgyűjtésre (Garbage Collection, GC).
Aktív hivatkozások száma: 2
Elérhető objektumok száma: 1
Elvetett objektumok száma: 1
Az első objektum, amelyre a **b** hivatkozott – az 1-es objektum – nem rendelkezik több hivatkozással, vagyis *elérhetetlen*.



```
c = null;
```

A `null` értéket a **c** változóhoz rendeled. Ez **c**-t *üres hivatkozássá* teszi, ami azt jelenti, hogy semmire nem hivatkozik. Ettől azonban még hivatkozó változó marad, és egy másik Book objektum továbbra is hozzárendelhető.
A 2-es objektumra továbbra is aktív hivatkozás (b) mutat, és amíg ez így marad, az objektum nem jelölhető ki szemétgyűjtésre.
Aktív hivatkozások száma: 1
Üres (*null*) hivatkozások száma: 1
Elérhető objektumok száma: 1
Elvetett objektumok száma: 1



A tömb olyan, mint egy csészékkel tele tálca

1 Bevezetsz egy int típusú tömbváltozót. A tömbváltozó egy tömbobjektum távirányítója.

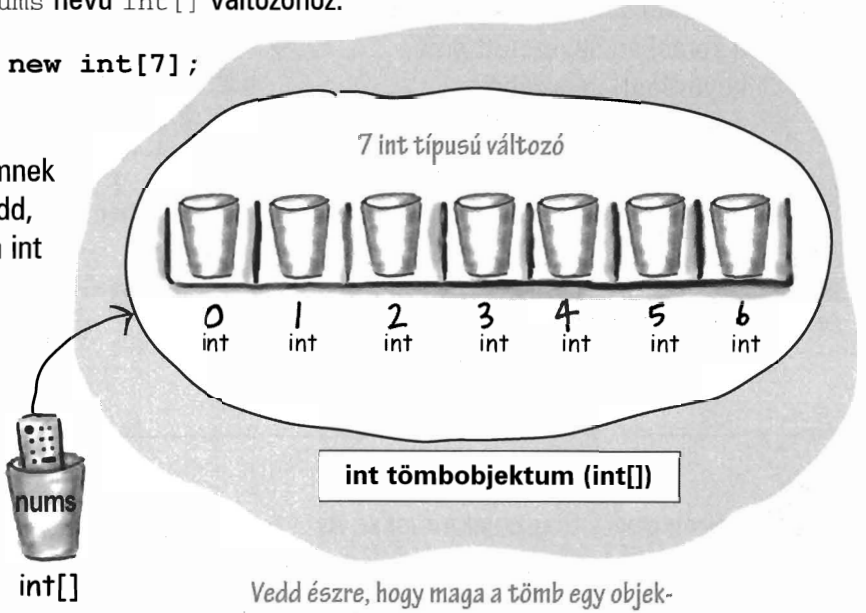
```
int[] nums;
```

2 Létrehozol egy új int tömböt, amelynek a hossza 7, és hozzárendeled a korábban bevezetett `nums` nevű `int[]` változóhoz:

```
nums = new int[7];
```

3 A tömbben levő minden elemnek adsz egy int értéket. Ne feledd, az int *tömbök* elemei csupán int *változók*.

```
nums[0] = 6;
nums[1] = 19;
nums[2] = 44;
nums[3] = 42;
nums[4] = 10;
nums[5] = 20;
nums[6] = 1;
```



Vedd észre, hogy maga a tömb egy objektum, bár a benne levő 7 elem primitív.

A tömbök is objektumok

A Java alapkönyvtára (standard library) egy csomó kifinomult adatszerkezetet tartalmaz, például térképeket (megfeleltetés, map), fákat (tree) és halmazokat (set; lásd a B függelékben), de ha csak egy gyors, rendezett, hatékony listára vágysz, a tömbök nyújtják a legkényelmesebb megoldást. A tömbökben gyorsan elérhetsz egy tetszőleges elemet, csak meg kell adnod a sorszámát, vagyis a helyét a tömbben.

A tömbökben minden elem csupán egy változó. Más szóval: egy a nyolc primitív változótípus (Large Furry Dog, emlékszel?) közül, vagy egy hivatkozó változó. Bármi, amit beletehetsz az adott típusú *változóba*, hozzárendelhető egy ugyanolyan típusú *tömbelemhez*. Egy int (int[]) típusú tömbben tehát minden elem egy egész (int) értéket tárol, egy Dog tömbben (Dog[]) pedig…egy Dog-ot? Nem. Emlékezz vissza, hogy a hivatkozó változók csak egy hivatkozást tárolnak (egy távirányítót), nem magát az objektumot. Egy Dog tömbben tehát az elemek egy távirányítót tárolhatnak egy Dog-ra. A Dog objektumokat természetesen létre kell hozni – ezt fogod látni a következő oldalon.

Figyelj meg egy kulcsfontosságú dolgot a fenti ábrán: a *tömb egy objektum, annak ellenére, hogy primitívek tömbjéből áll.*

A tömbök mindig objektumok, akár úgy vezetted be őket, hogy primitíveket, akár úgy, hogy objektumhivatkozásokat tároljanak. Lehet viszont olyan tömbobjektumod, amely a bevezetése szerint primitív értékeket *tárol*. Más szavakkal, a tömbobjektumnak lehetnek primitív *elemei*, de maga a tömb *soha nem* primitív. Függetlenül attól, hogy mit tárol a tömb, maga a tömb mindig objektum!

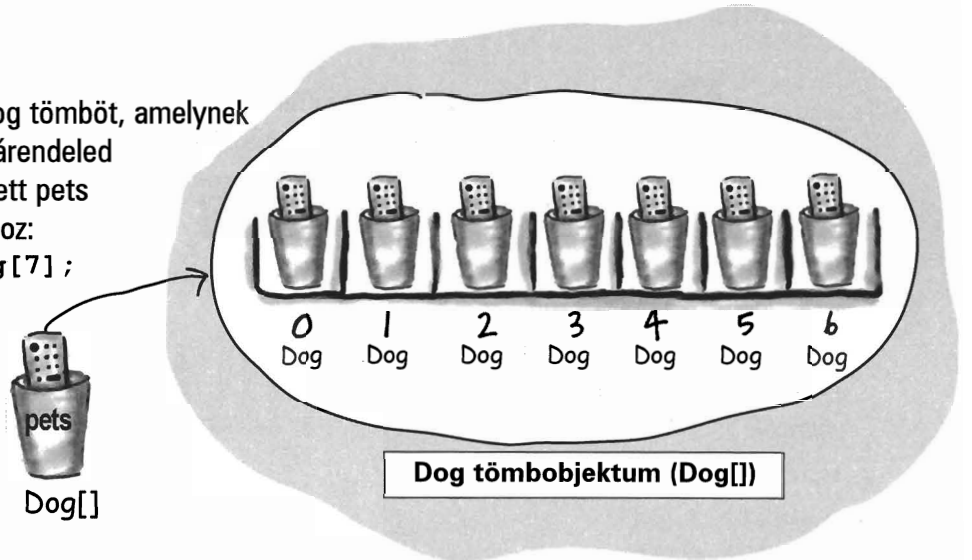
Dog objektumokból álló tömb létrehozása

1 Bevezetsz egy Dog típusú tömbváltozót:

```
Dog[] pets;
```

2 Létrehozol egy új Dog tömböt, amelynek a hossza 7, és hozzárendeled a korábban bevezetett pets nevű Dog[] változóhoz:

```
pets = new Dog[7];
```



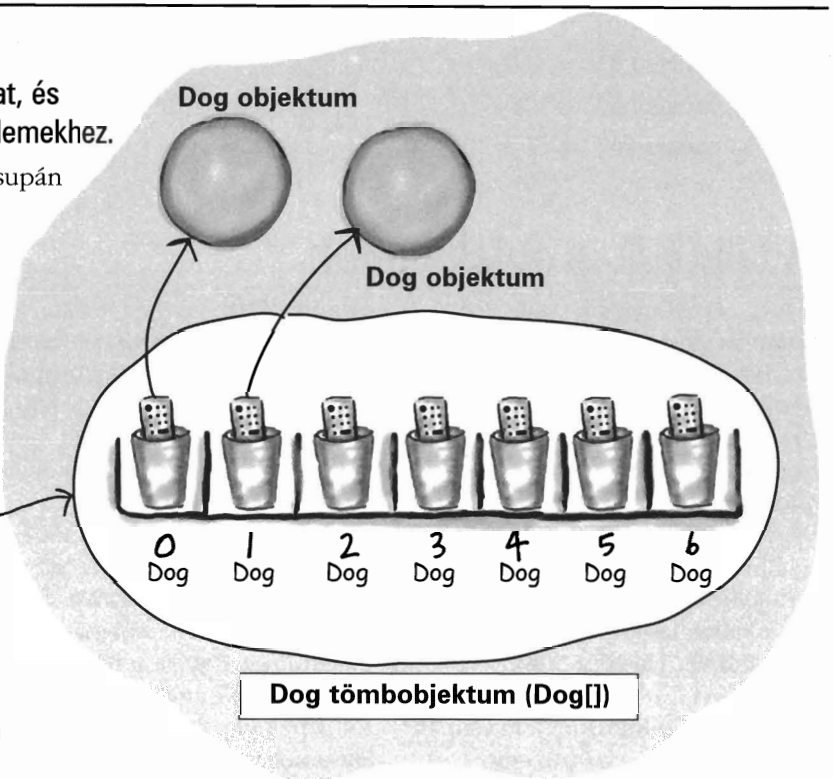
Mi hiányzik?

A Kutyák! Van egy Dog *hivatkozásokból* álló tömböd, de nincsenek Dog *objektumaid*!

3 Létrehozod a Dog objektumokat, és hozzárendeled azokat a tömbelemekhez.

Ne feledd, a Dog *tömb* elemei csupán *hivatkozó változók*. Továbbra is szükség van Dog-okra!

```
pets[0] = new Dog();
pets[1] = new Dog();
```



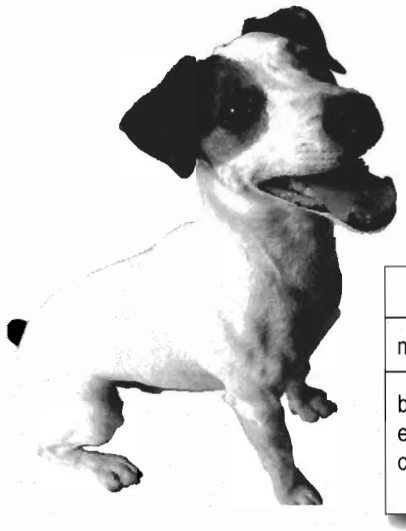
Hegyezd ki a ceruzád!

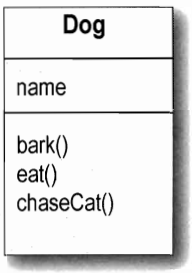
Mi a pets[2] jelenlegi értéke?

_____

Milyen kóddal érheted el, hogy a pets[3] a két létező Dog objektum egyikére hivatkozzon?

_____





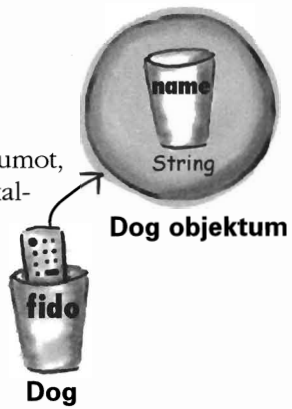
Vezéreld a Kutyád!

(egy hivatkozó változóval)

```
Dog fido = new Dog();

fido.name = "Fido";
```

Itt létrehoztunk egy Dog objektumot, és a ***fido*** hivatkozó változóra alkalmazott pont művelettel elértük a name változót* A fido hivatkozás segítségével ráveheted a kutyát, hogy ugasson, egyen vagy megkergesse a macskát (bark(), eat(), chaseCat()):




```
fido.bark();

fido.chaseCat();
```

A Java odafigyel a típusokra.

Ha egy tömböt már bevezettél, semmi mást nem tehetsz bele, csak a bevezetésben megadott tömbtípusnak megfelelő dolgokat.

Egy Kutya (Dog) tömbbe például nem tehetsz egy Macskát (Cat) – elég szörnyű lenne, ha valaki azt hinné, hogy csak Kutyák vannak a tömbben, és csak akkor venné észre rémülten, hogy egy macska is lapít közöttük, amikor kiadná a parancsot az ugatásra. Ugyanígy nem gyömöszölhetsz egy double értéket sem egy int tömbbe (mert kicsordul, emlékszel?). Egy byte viszont nyugodtan mehet ugyanoda, mert egy byte érték mindig elfér egy int méretű csészében. Ezt hívják **beleértett (implicit), közvetett vagy rejtett bővítésnek.** **A részleteket később majd tisztázzuk; most elég annyit tudnod, hogy a tömb bejelentett típusát észben tartva a fordító nem engedi, hogy egy tömbbe nem oda való dolgot tegyél.**

Mi történik, ha a Dog egy Dog tömbben található?

Azt már tudod, hogy a pont művelettel érheted el a Dog példányváltozóit és tagfüggvényeit – de *mire* kell alkalmaznod?

Ha a Dog egy tömbben található, akkor nincs egy olyan igazi változóneved, mint a ***fido***. Ehelyett a tömbjelölést kell használnod, és a tömb egy adott helyén (index, pozíció) található objektumra kell megnyomnod a távirányító gombját (a pont műveletet):

```
Dog[] myDogs = new Dog[3];

myDogs[0] = new Dog();

myDogs[0].name = "Fido";

myDogs[0].bark();
```

* Igen, tudjuk, hogy itt nincs betokozás (egységbe zárás), de próbáltunk egyszerűek maradni – egyelőre. A betokozásról a 4. fejezetben lesz szó.

```
class Dog {
  String name;
  public static void main (String[] args) {
    // Dog objektum létrehozása és elérése
    Dog dog1 = new Dog();
    dog1.bark();
    dog1.name = "Bart";

    // most létrehozol egy Dog tömböt
    Dog[] myDogs = new Dog[3];
    // és beleteszel néhány kutyát
    myDogs[0] = new Dog();
    myDogs[1] = new Dog();
    myDogs[2] = dog1;

    // a Kutyák elérése a tömbhivatkozások
    // segítségével
    myDogs[0].name = "Fred";
    myDogs[1].name = "Marge";

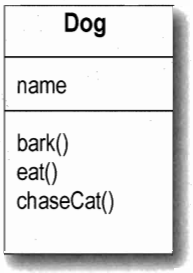
    //Hm... Mi a myDogs[2] neve?
    System.out.print("last dog's name is ");
    System.out.println(myDogs[2].name);

    // most végigmész a tömbön, és
    // ugatásra utasítasz minden kutyát
    int x = 0;
    while(x < myDogs.length) {
      myDogs[x].bark();
      x = x + 1;
    }
  }

  public void bark() {
    System.out.println(name + " says Ruff!");
  }
  public void eat() { }
  public void chaseCat() { }
}
```

a tömböknek mindig van egy „length" változójuk, ami megadja neked a tömbben levő elemek számát

Kutyás példa



Kimenet:

```
File Edit Window Help Howl
%java Dog
null says Ruff!
last dog's name is Bart
Fred says Ruff!
Marge says Ruff!
Bart says Ruff!
```

Újratöltve

- A változóknak két fajtája van: primitívek és hivatkozások.
- A változókat mindig egy névvel és egy típussal kell bevezetned.
- A primitív változók értéke az értéket ábrázoló biteket (5, 'a', true, 3.1416 stb.) tárolja.
- A hivatkozó változók értéke olyan biteket tárol, amelyek azt írják le, hogy miként lehet elérni az adott objektumot a kupacban.
- A hivatkozó változók olyanok, mint egy távirányító. Ha a pont műveletet (.) egy hivatkozó változón hajtod végre, az olyan, mintha megnyomnál egy gombot a távirányítón, hogy elérj egy tagfüggvényt vagy példányváltozót.
- A hivatkozó változók értéke null, ha egyetlen objektumra sem hivatkoznak.
- A tömbök mindig objektumok, még ha a tömböt úgy vezetted is be, hogy primitíveket tároljon. Nem létezik olyan, hogy primitív tömb, csak olyan tömb, amely primitíveket *tárol*.



Gyakorlat

Légy a fordító!



Az ezen az oldalon található Java-fájlok mindegyike egy-egy teljes forrásfájl. A te feladatod az, hogy eljátszd a fordítóprogram szerepét, és kiderítsd, hogy lefordíthatók-e ezek a fájlok. Ha nem, hogyan javítanád ki őket?

A

```
class Books {
  String title;
  String author;
}

class BooksTestDrive {
  public static void main(String [] args) {

    Books [] myBooks = new Books[3];
    int x = 0;
    myBooks[0].title = "The Grapes of Java";
    myBooks[1].title = "The Java Gatsby";
    myBooks[2].title = "The Java Cookbook";
    myBooks[0].author = "bob";
    myBooks[1].author = "sue";
    myBooks[2].author = "ian";

    while (x < 3) {
      System.out.print(myBooks[x].title);
      System.out.print(" by ");
      System.out.println(myBooks[x].author);
      x = x + 1;
    }
  }
}
```

B

```
class Hobbits {

  String name;

  public static void main(String [] args) {

    Hobbits [] h = new Hobbits[3];
    int z = 0;

    while (z < 4) {
      z = z + 1;
      h[z] = new Hobbits();
      h[z].name = "bilbo";
      if (z == 1) {
        h[z].name = "frodo";
      }
      if (z == 2) {
        h[z].name = "sam";
      }
      System.out.print(h[z].name + " is a ");
      System.out.println("good Hobbit name");
    }
  }
}
```



Gyakorlat

Kódmágnesek

Egy Java-program kódja összekeveredett a hűtőmágneseken. Össze tudod rakni úgy a kódrészleteket, hogy előálljon belőlük a lap alján látható kimenetet eredményező Java-program? Néhány kapcsos zárójel leesett a padlóra, és túl kicsi volt ahhoz, hogy fel tudjuk venni, úgyhogy nyugodtan egészítsd ki a kódot annyi kapcsos zárójellel, amennyit csak szükségesnek érzel!



```
int y = 0;
```

```
ref = index[y];
```

```
islands[0] = "Bermuda";
islands[1] = "Fiji";
islands[2] = "Azores";
islands[3] = "Cozumel";
```

```
int ref;
while (y < 4) {
```

```
System.out.println(islands[ref]);
```

```
index[0] = 1;
index[1] = 3;
index[2] = 0;
index[3] = 2;
```

```
String [] islands = new String[4];
```

```
System.out.print("island = ");
```

```
int [] index = new int[4];
```

```
y = y + 1;
```

```
File Edit Window Help Bikini
% java TestArrays
island = Fiji
island = Cozumel
island = Bermuda
island = Azores
```

```
class TestArrays {

  public static void main(String [] args) {
```



Halászd ki a medencéből!



A ***feladatod*** az, hogy kihalászd a kódda-rabokat a medencéből, és elhe-lyezd azokat a kód üresen ha-gyott helyein. Egy kóddarabot egynél **többször** is felhasznál-hatsz, viszont nem lesz szükséged minden darabra. A ***cél*** egy olyan osztály felépítése, amely lefordítható, futtatható, és az alább látható kimenetet eredményezi.

Kimenet:

```
File  Edit  Window  Help  Bermuda
%java Triangle
triangle 0, area = 4.0
triangle 1, area = 10.0
triangle 2, area = 18.0
triangle 3, area = ____
y = _____________________
```

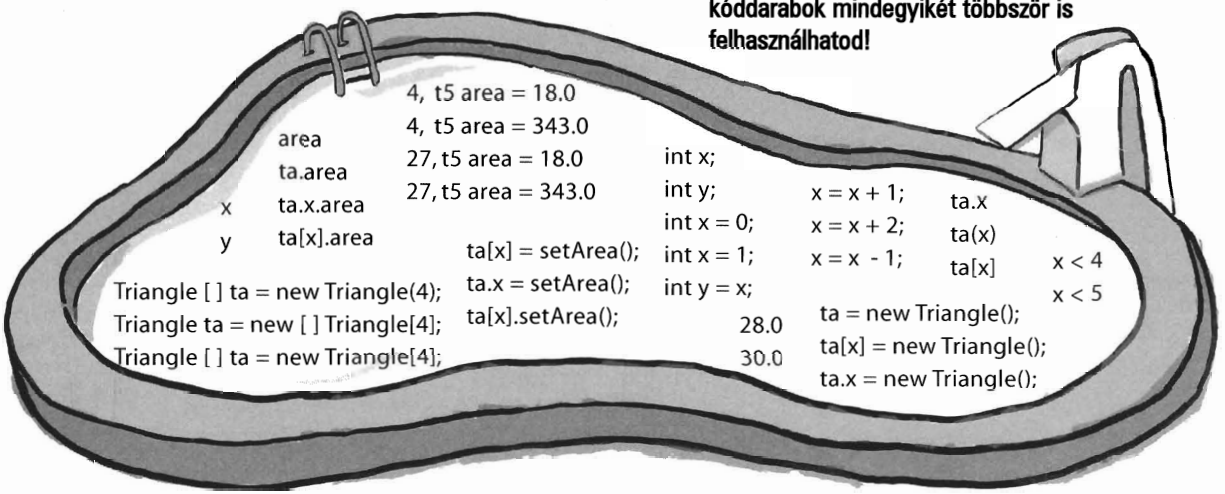
Bónusz kérdés!

Bónuszpontokért halászd ki azokat a kóddarabokat a medencéből, amelyekkel kipótolható a fenti kimenet hiányzó része!

```
class Triangle {
  double area;
  int height;
  int length;
  public static void main(String [] args) {
    ___________
    ___________________________
    while ( __________ ) {
      ___________________________
      ________.height = (x + 1) * 2;
      ________.length = x + 4;
      ___________________________
      System.out.print("triangle "+x+", area");
      System.out.println(" = " + ________.area);
      ________________
    }
    ________________
    x = 27;
    Triangle t5 = ta[2];
    ta[2].area = 343;
    System.out.print("y = " + y);
    System.out.println(", t5 area = "+ t5.area);
  }
  void setArea() {
    ____________ = (height * length) / 2;
  }
}
```

(Néha nem használunk külön tesztosztályt, mert takarékos-kodni szeretnénk a hellyel a la-pon.)

Megjegyzés: A medencében úszó kóddarabok mindegyikét többször is felhasználhatod!





Túrd fel a kupacot!

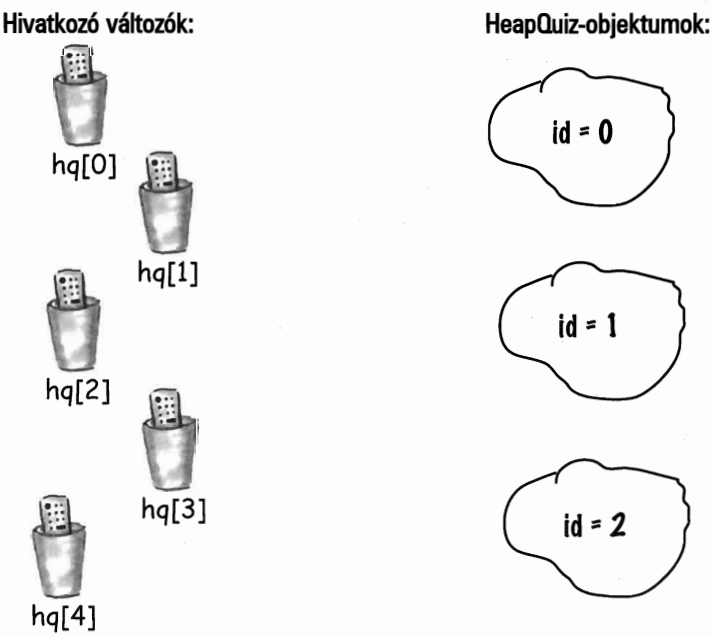
A jobb oldalon egy rövid Java-programot láthatsz. Amikor a program eléri a „//csinálunk valamit" részt, létrejönnek bizonyos objektumok és hivatkozó változók. A feladatod az, hogy kideríts, mely objektumokra mutatnak az egyes hivatkozások. Nem mindegyik hivatkozó változót kell felhasználnod, és egyes objektumokra több hivatkozás is mutathat. Kösd össze a hivatkozásokat a nekik megfelelő objektumokkal!

Tipp: Hacsak nem vagy nálunk sokkal ügyesebb, valószínűleg rajzolnod kell majd néhány olyan diagramot, mint amilyeneket pár oldallal korábban láttál. Használj ceruzát, hogy berajzolás után kiradírozhasd a hivatkozási kapcsolatokat (a hivatkozó távirányítóktól az objektumokhoz vezető nyilakat).

```
class HeapQuiz {
  int id = 0;
  public static void main(String [] args) {
    int x = 0;
    HeapQuiz [ ] hq = new HeapQuiz[ 5] ;
    while ( x < 3 ) {
      hq[ x] = new HeapQuiz();
      hq[ x] .id = x;
      x = x + 1;
    }
    hq[ 3] = hq[ 1] ;
    hq[ 4] = hq[ 1] ;
    hq[ 3] = null;
    hq[ 4] = hq[ 0] ;
    hq[ 0] = hq[ 3] ;
    hq[ 3] = hq[ 2] ;
    hq[ 2] = hq[ 0] ;
    // csinálunk valamit
  }
}
```

Köss össze minden hivatkozást a neki megfelelő objektummal (objektumokkal)!

Nem biztos, hogy minden hivatkozásra szükség lesz.





Az elcsent hivatkozások esete

Sötét, viharos éjszaka volt. Tawny úgy sétált be a programozók főhadiszállására, mintha az övé lett volna az épület. Tudta, hogy még minden programozó bőszen dolgozik, és segítségre volt szüksége. Egy új tagfüggvényt kellett adnia az ügyfél új, szigorúan titkos, Java-képes mobiltelefonjára betöltendő központi osztályhoz. A mobil memóriáját olyan szűkre szabták, mint Tawny felsőjét, és ezt mindenki tudta. A programozók farkasvermének szokásos nyüzsgését fagyos csend váltotta fel, ahogy Tawny utat tört magának a fehér táblához. Gyorsan felrajzolta az új tagfüggvény képességeinek vázlatát, és lassan körbetekintett a teremben. „Nos, fiúk, ideje, hogy megmutassátok, mit tudtok…" – dorombolta – „Aki a leginkább memóriatakarékos változatot készíti ebből a tagfüggvényből, az jöhet velem holnap az ügyfél nyitóbulijára Maui-ra…hogy segítsen telepíteni az új szoftvert."

Ötperces rejtély



Másnap reggel Tawny belibbent a főhadiszállásra rövid kis Aloha-dresszében. „Uraim," – mosolygott – „a gép néhány óra múlva indul – mutassák, mijük van!" Bob jelentkezett elsőként, de amint nekilátott felvázolni a tervét a táblára, Tawny félbeszakította: „Térjünk a lényegre, Bob! Mutasd meg, hogyan kezelted a kapcsolati objektumok frissítését!" Bob gyorsan felfirkantotta a kérdéses kódrészt a táblára:

```
Contact [] ca = new Contact[ 10] ;
while ( x < 10 ) {          // 10 kapcsolati objektum létrehozása
  ca[ x] = new Contact();
  x = x + 1;
}
  // bonyolult kapcsolatilista-frissítési műveletek a ca-val
```

„Tawny, tudom, hogy a memória szűkös, de a leírásod szerint mind a tíz lehetséges kapcsolat adatait tudnunk kell elérni, ezért ennél jobb megoldást nem tudtam kifőzni." – magyarázta Bob. Kent következett, aki már látta maga előtt, ahogy kókuszkoktélt szürcsöl Tawny-val. „Bob," – sóhajtott – „a te megoldásod csikorog egy kissé, nem gondolod?" Majd széles vigyorral folytatta: „Ezt figyeld, bébi:"

```
Contact refc;
while ( x < 10 ) {          // 10 kapcsolati objektum létrehozása
  refc = new Contact();
  x = x + 1;
}
  // bonyolult kapcsolatilista-frissítési műveletek a ca-val
```

„Én egy csomó hivatkozást megspóroltam, hogy kíméljem a memóriát, Bobby fiú. Elteheted a naptejet." – gúnyolódott Kent. „Lassan a testtel, Kent!" – állította meg Tawny – „Lehet, hogy megmentettél egy kis memóriát, de Bob az, aki velem jön!"

Miért választotta Tawny Bob tagfüggvényét Kentével szemben, amikor Kent kevesebb memóriát használt?



Gyakorlat – Megoldás

Kódmágnesek:

```
class TestArrays {
  public static void main(String [] args) {
    int [] index = new int[4];
    index[0] = 1;
    index[1] = 3;
    index[2] = 0;
    index[3] = 2;
    String [] islands = new String[4];
    islands[0] = "Bermuda";
    islands[1] = "Fiji";
    islands[2] = "Azores";
    islands[3] = "Cozumel";
    int y = 0;
    int ref;
    while (y < 4) {
      ref = index[y];
      System.out.print("island = ");
      System.out.println(islands[ref]);
      y = y + 1;
    }
  }
}
```

```
File Edit Window Help Bikini
% java TestArrays
island = Fiji
island = Cozumel
island = Bermuda
island = Azores
```

```
class Books {
  String title;
  String author;
}
class BooksTestDrive {
  public static void main(String [] args) {
    Books [] myBooks = new Books[3];
    int x = 0;
A   myBooks[0] = new Books();
    myBooks[1] = new Books();
    myBooks[2] = new Books();
    myBooks[0].title = "The Grapes of Java";
    myBooks[1].title = "The Java Gatsby";
    myBooks[2].title = "The Java Cookbook";
    myBooks[0].author = "bob";
    myBooks[1].author = "sue";
    myBooks[2].author = "ian";
    while (x < 3) {
      System.out.print(myBooks[x].title);
      System.out.print(" by ");
      System.out.println(myBooks[x].author);
      x = x + 1;
    }
  }
}
```

Ne feledd: a Books objektumokat még létre kell hozni!

```
class Hobbits {
  String name;
  public static void main(String [] args) {
    Hobbits [] h = new Hobbits[3];
    int z = -1;
    while (z < 2) {
      z = z + 1;
      h[z] = new Hobbits();
B     h[z].name = "bilbo";
      if (z == 1) {
        h[z].name = "frodo";
      }
      if (z == 2) {
        h[z].name = "sam";
      }
      System.out.print(h[z].name + " is a ");
      System.out.println("good Hobbit name");
    }
  }
}
```

Ne feledd: a tömbök a 0. elemmel kezdődnek!



Fejtörők – Megoldás

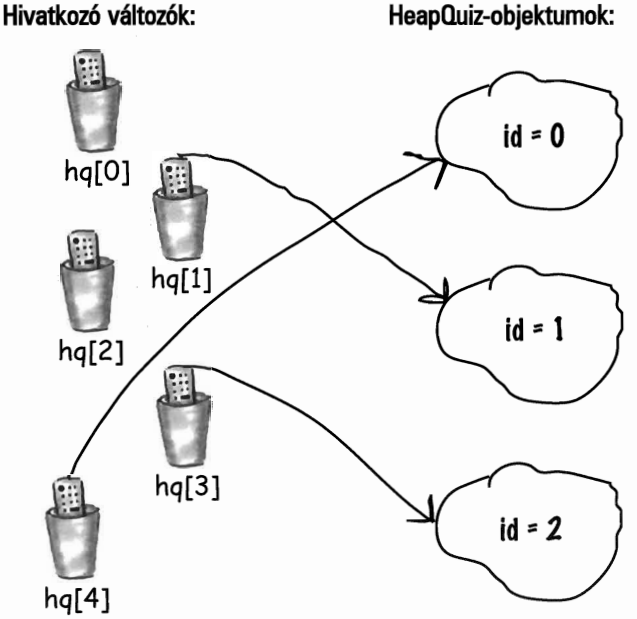
```
class Triangle {
  double area;
  int height;
  int length;
  public static void main(String [] args) {
    int x = 0;
    Triangle [ ] ta = new Triangle[4];
    while ( x < 4 ) {
      ta[x] = new Triangle();
      ta[x].height = (x + 1) * 2;
      ta[x].length = x + 4;
      ta[x].setArea();
      System.out.print("triangle "+x+", area");
      System.out.println(" = " + ta[x].area);
      x = x + 1;
    }
    int y = x;
    x = 27;
    Triangle t5 = ta[2];
    ta[2].area = 343;
    System.out.print("y = " + y);
    System.out.println(", t5 area = "+ t5.area);
  }
  void setArea() {
    area = (height * length) / 2;
  }
}
```

```
File Edit Window Help Bermuda
%java Triangle
triangle 0, area = 4.0
triangle 1, area = 10.0
triangle 2, area = 18.0
triangle 3, area = 28.0
y = 4, t5 area = 343
```

Az elcsent hivatkozások esete

Tawny látta, hogy Kent tagfüggvényében van egy komoly hiba. Igaz, hogy nem használt annyi hivatkozó változót, mint Bob, de semmilyen módon nem lehetett elérni egyetlen Contact objektumot sem a tagfüggvény által létrehozott utolsó kivételével. A ciklus minden lefutásakor új objektumot rendelt az egyetlen hivatkozó változóhoz, így a korábban hivatkozott objektumokat sorban a kupacra hajította – *elérhetetlenül*. Mivel a tíz létrehozott objektumból kilenchez nem lehetett hozzáférni, Kent tagfüggvényének nem volt semmi haszna.

(A szoftver hatalmas sikert aratott, ezért az ügyfél fizetett még egy hetet Tawny-nak és Bobnak Hawaii-n. Itt szeretnénk megjegyezni, hogy ha végigolvasod a könyvet, te is hasonlókra számíthatsz.)



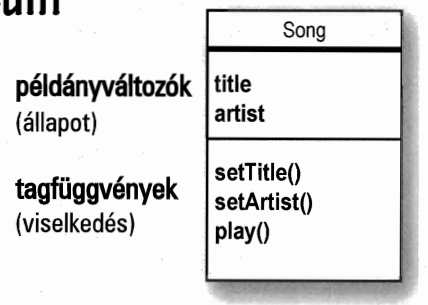
Hogyan viselkednek az objektumok?



Az állapot befolyásolja a viselkedést, a viselkedés pedig hatással van az állapotra. Már tudod, hogy az objektumoknak állapota (state) és viselkedése (behavior) is van, amelyeket **példányváltozók** és **tagfüggvények** ábrázolnak. Eddig azonban arról nem ejtettünk szót, hogy az állapot és a viselkedés milyen *kapcsolatban* áll egymással. Azt már tudod, hogy egy osztály minden példánya (vagyis minden objektum, amely az adott típusba tartozik), saját, egyedi értékeket tárol a példányváltozóiban. A Dog A neve (*name*) lehet „Fido", a súlya (*weight*) pedig 35 kiló, míg a Dog B a „Killer" nevet viselheti a 4,5 kilójához. Tegyük fel, hogy a Dog osztálynak van egy makeNoise() – csinalZaj() – tagfüggvénye. Szerinted egy 35 kilós eb nem ugat kissé mélyebben, mint egy 4,5 kilós? (Már ha az a cincogásszerű vakkantás *ugatásnak* nevezhető.) Szerencsére éppen ez az objektumok lényege: *viselkedésük* van, vagyis olyan műveleteik, amelyeket az objektum *állapotán* hajthatunk végre. Más szavakkal, a ***tagfüggvények példányváltozó-értékeket használnak***: „ha a kutya könnyebb 7 kilónál, vakkants fejhangon, máskülönben…" vagy „növeld a súlyt 5 kilóval". Gyerünk, változtassunk ezen az állapoton!

Ne feledd: az osztályok azt írják le, hogy mit tud és mit csinál egy objektum

Minden osztály egy objektum tervrajza. Amikor osztályt írsz, azt írod le, hogyan hozhat létre a JVM egy adott típusú objektumot. Azt már tudod, hogy minden adott típusú objektum más-más *példányváltozó*-értékekkel rendelkezhet. De mi a helyzet a tagfüggvényekkel?

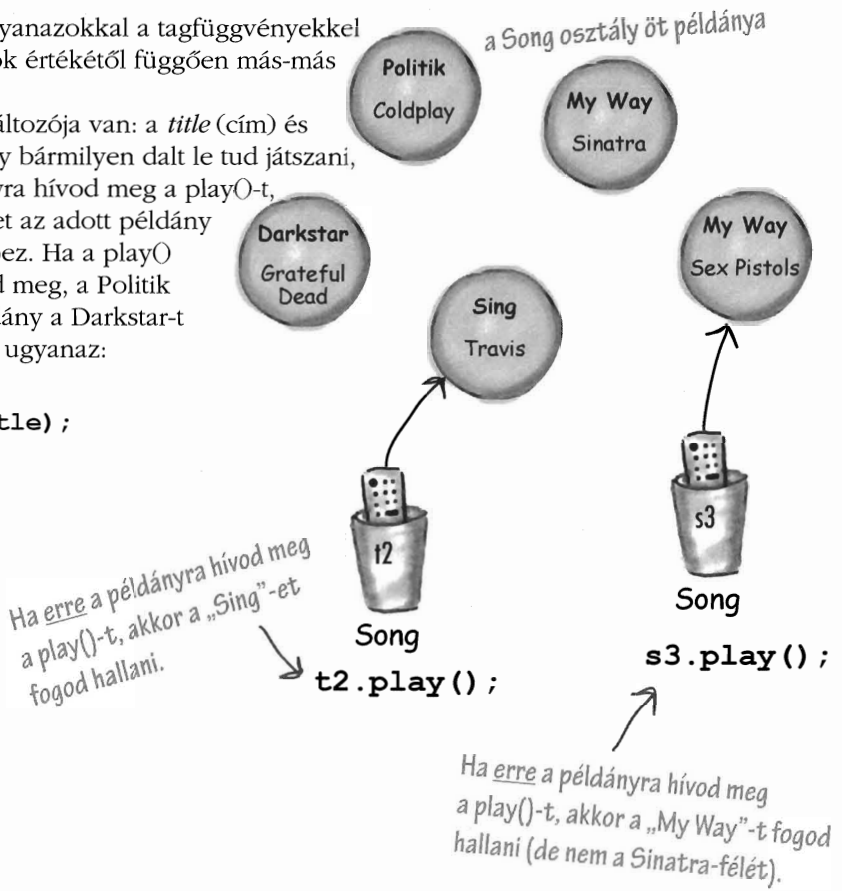


Viselkedhetnek másképp a tagfüggvények minden adott típusú objektumban?

Hát…igen is, meg nem is.*

Egy adott osztály minden példánya ugyanazokkal a tagfüggvényekkel rendelkezik, de azok a példányváltozók értékétől függően más-más módon *viselkedhetnek*.
A Song (Dal) osztálynak két példányváltozója van: a *title* (cím) és az *artist* (előadó). A play() tagfüggvény bármilyen dalt le tud játszani, de attól függően, hogy melyik példányra hívod meg a play()-t, más-más dalt fogsz hallani: azt, amelyet az adott példány *title* példányváltozójának értéke jelképez. Ha a play() tagfüggvényt az egyik példányra hívod meg, a Politik című dal csendül fel, míg a másik példány a Darkstar-t indítja el. A tagfüggvény kódja viszont ugyanaz:

```
void play() {
    soundPlayer.playSound(title);
}
```

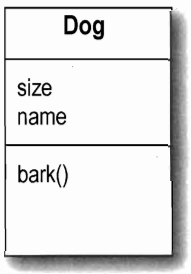



```
Song t2 = new Song();
t2.setArtist("Travis");
t2.setTitle("Sing");
Song s3 = new Song();
s3.setArtist("Sex Pistols");
s3.setTitle("My Way");
```

* Igen, már megint egy elképesztően egyértelmű válasz!

A méret befolyásolja az ugatást

Egy kis Kutya (Dog) másképp ugat, mint egy nagy Kutya. A Dog osztálynak van egy *size* (méret) nevű példányváltozója, amelyet a *bark()* – ugat() – tagfüggvény arra használ, hogy eldöntse, milyen hangon kell ugatnia.




```
class Dog {
  int size;
  String name;

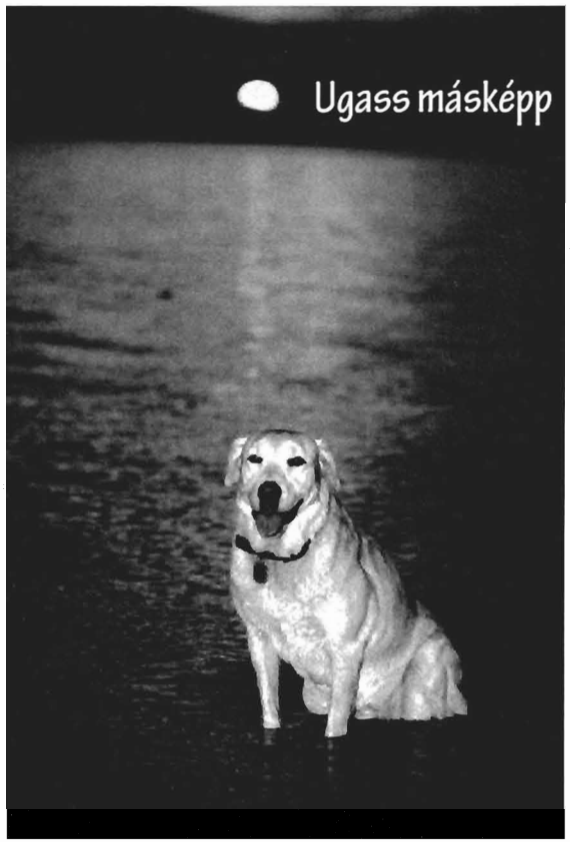
  void bark() {
    if (size > 60) {
       System.out.println("Wooof! Wooof!");
    } else if (size > 14) {
       System.out.println("Ruff!  Ruff!");
    } else {
       System.out.println("Yip! Yip!");
    }
  }
}
```

```
class DogTestDrive {

   public static void main (String[] args) {
     Dog one = new Dog();
     one.size = 70;
     Dog two = new Dog();
     two.size = 8;
     Dog three = new Dog();
     three.size = 35;

     one.bark();
     two.bark();
     three.bark();
   }
}
```

```
File Edit Window Help Playdead
%java DogTestDrive
Wooof! Wooof!
Yip! Yip!
Ruff!  Ruff!
```



A tagfüggvényeknek átadhatsz dolgokat

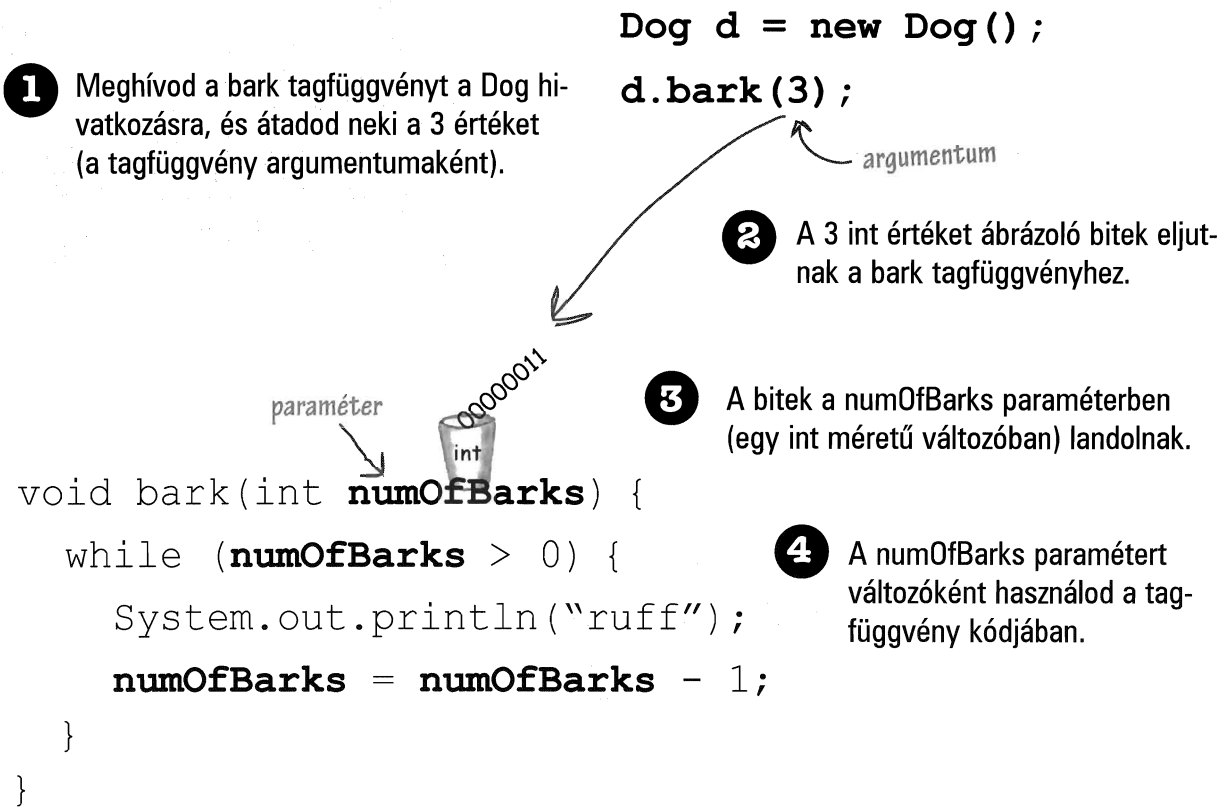
Ahogy ezt egy programozási nyelvtől el is várhatod, a tagfüggvényeidnek a Javában is átadhatsz értékeket. Előfordulhat például, hogy meg szeretnéd mondani egy Dog objektumnak, hogy hányszor ugasson. Ezt ezzel a hívással teheted meg:

```
d.bark(3);
```

A programozói hátteredtől és a személyes ízlésedtől függően a tagfüggvényeknek átadott értékeket hívhatod *argumentumoknak* és *paramétereknek* is. Bár a számítógéptudományban azok a laborban dolgozó köpenyesek, akik szinte biztosan nem olvassák ezt a könyvet, hivatalosan *különbséget* tesznek a kettő között, nekünk nagyobb cél lebeg a szemünk előtt. Szóval *te* úgy hívod őket, ahogy akarod (argumentum, fánk, szőrgolyó stb.), mi viszont ehhez ragaszkodunk:

A tagfüggvények paramétereket használnak. A hívók argumentumokat adnak át.

A tagfüggvényeknek átadott dolgokat tehát argumentumoknak fogjuk hívni. Az ***argumentum*** (vagyis egy olyan érték, mint a 2, a „Foo" vagy egy Dog-ra mutató hivatkozás) pedig nem máshol landol, mint egy…felkészülni…***paraméterben***. A paraméterek csupán helyi változók – olyan, típussal és névvel rendelkező változók, amelyeket egy tagfüggvény törzsén belül használhatsz. A lényeg azonban a következő: **ha egy tagfüggvény vár valamilyen paramétert, akkor *muszáj* átadnod neki valamit**. Annak a valaminek pedig egy megfelelő típusú értéknek kell lennie.



A tagfüggvényektől *vissza* is kaphatsz dolgokat

A tagfüggvények vissza is adhatnak értékeket. Minden tagfüggvény bevezetésekor meg kell adni egy visszatérési típust, de eddig minden tagfüggvényünk visszaadott típusát **void**-ként határoztuk meg, ami azt jelenti, hogy nem adtak vissza semmit.

```
void go() {

}
```

Egy tagfüggvényt azonban bevezethetsz úgy is, hogy egy adott típusú értéket adjon vissza a hívónak. Például:

```
int giveSecret() {
   return 42;
}
```

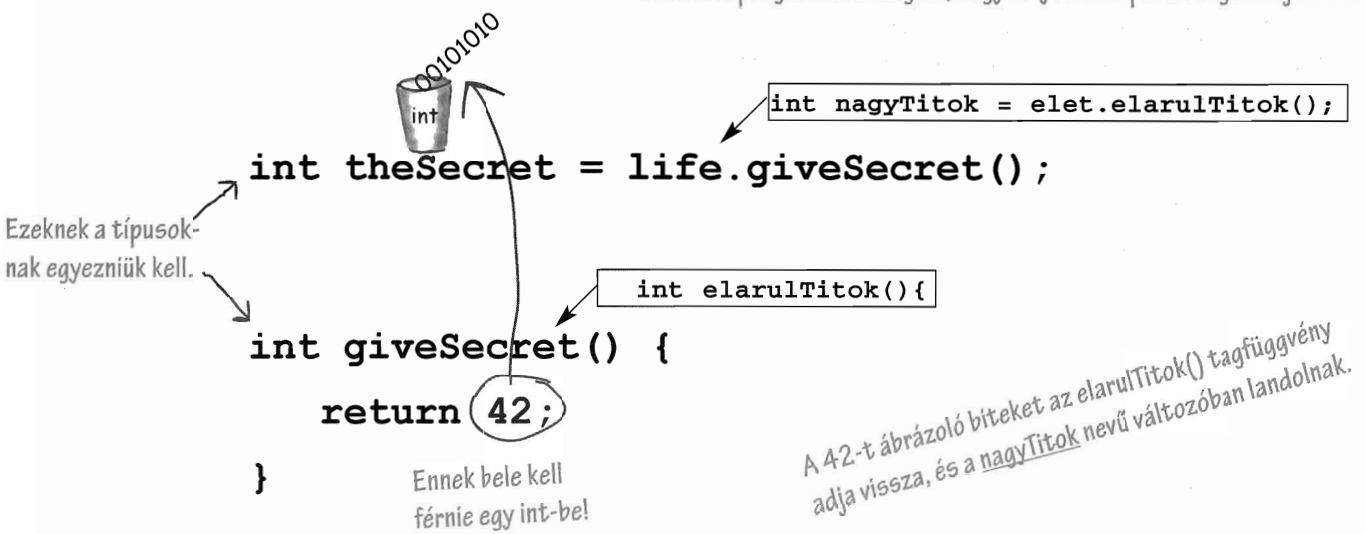
```
int elarulTitok(){
```

Ha egy tagfüggvényt úgy vezetsz be, hogy visszaadjon egy értéket, akkor *kötelező* a bejelentett típusú értéket visszaadnod. (De legalábbis egy a bejelentett típussal *összeegyeztethető* – kompatibilis – értéket. Erről bővebben is szót ejtünk, amikor a többalakúságot tárgyaljuk a 7. és 8. fejezetben.)



Azt add vissza, amit ígértél!

A fordítóprogram nem engedi, hogy helytelen típusú dolgot adj vissza.



Egy tagfüggvénynek több dolgot is átadhatsz

A tagfüggvényeknek több paraméterük is lehet. Bevezetéskor vesszővel kell elválasztanod őket, akárcsak átadáskor az argumentumokat. Ami pedig a legfontosabb: ha egy tagfüggvénynek paraméterei vannak, akkor a megfelelő típusú argumentumokat *kell* átadnod, a megfelelő sorrendben.

Kétparaméteres tagfüggvény meghívása és két argumentum átadása:

```
void go() {
    TestStuff t = new TestStuff();
    t.takeTwo(12, 34);
}

void takeTwo(int x, int y) {
    int z = x + y;
    System.out.println("Total is " + z);
}
```

Az átadott argumentumok ugyanabban a sorrendben érkeznek, ahogy átadtad őket. Az első argumentum az első paraméterbe kerül, a második a másodikba, és így tovább.

Egy tagfüggvénynek átadhatsz változókat is, feltéve, hogy azok típusa megegyezik a paraméter típusával:

```
void go() {
    int foo = 7;
    int bar = 3;
    t.takeTwo(foo, bar);
}

void takeTwo(int x, int y) {
    int z = x + y;
    System.out.println("Total is " + z);
}
```

A foo és a bar értéke az x és y paraméterekben landol. Az x-ben levő bitek tehát most megegyeznek a foo bitjeivel (a „7" egész szám bitmintájával), az y-ban levők pedig a bar bitjeivel.

Mi a z értéke? Ugyanaz, mint amit akkor kapnál, ha a foo-t és a bar-t összeadnád, mielőtt átadnád a takeTwo tagfüggvénynek.

A Java érték szerinti átadást alkalmaz
Ez azt jelenti, hogy másolatokat ad át



```
int x = 7;
```



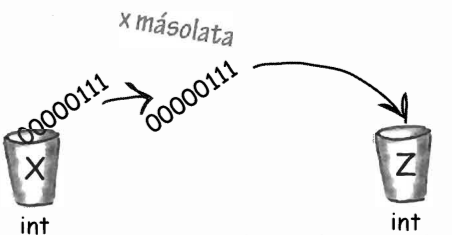


1 Bevezetsz egy int típusú változót, és hozzárendeled a „7" értéket. Az x nevű változóba a 7 bitmintája kerül.

```
void go(int z){ }
```



2 Bevezetsz egy z nevű int paraméterrel ellátott tagfüggvényt.




```
foo.go(x);
```

```
void go(int z){ }
```

3 Meghívod a go() tagfüggvényt, argumentumként átadva neki az x változót. A Java lemásolja az x-ben levő biteket, és ez a másolat landol a z-ben.




```
void go(int z){
    z = 0;
}
```

4 Módosítod a z értékét a tagfüggvényen belül. Az x értéke nem változik, mert a z paraméternek átadott argumentum csak egy másolata volt az x-nek.

A tagfüggvény nem képes megváltoztatni a hívó x változóban levő biteket.

Nincsenek

hülye kérdések

K: Mi történik, ha az átadni kívánt argumentum nem primitív, hanem objektum?

V: Erről bővebben is szó lesz a későbbi fejezetekben, de már *tudod* a választ. A Java *mindent* érték szerint ad át. ***Mindent.*** Az *érték* pedig azt jelenti, hogy *„bitek egy változóban”*. Emlékezz vissza: nem objektumokat tömködsz a változókba, mert a változó csak egy távirányító – *egy objektumra mutató hivatkozás*. Tehát ha egy objektumra mutató hivatkozást adsz át egy tagfüggvénynek, akkor a *távirányító másolatát* adod át. De maradj velünk, mert erről lesz még mondanivalónk.

K: Egy tagfüggvény bevezetésekor megadhatok több visszatérési értéket is? Vagy van valamilyen mód rá, hogy egynél több értéket adjak vissza?

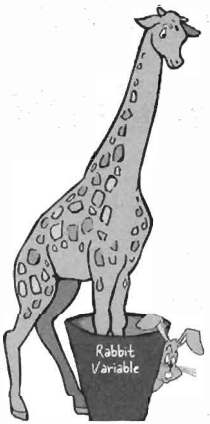
V: Mondjuk. Egy tagfüggvény csak egy visszatérési értéket vezethet be. DE...ha – teszem azt – három int értéket szeretnél visszaaadni, akkor visszatérési típusként megadhatsz egy int *tömböt*. Csak bele kell tömködnöd az inteket a tömbbe, és máris visszaadhatod őket. Ha a visszatérési értékek típusa különbözik, az már egy kicsit bonyolultabb – erre még visszatérünk egy későbbi fejezetben, amikor az ArrayList-ről beszélünk.

K: Pontosan azt a típust kell visszaadnom, amit bevezettem?

V: Bármit visszaadhatsz, ami *közvetetten* előléptethető az említett típusúvá, vagyis ahol egy int-et várnak, ott átadhatsz például egy byte-ot is. A hívót ez nem érdekli, mert a byte kényelmesen belefér az int-be, amit a hívó az eredmény tárolására használ. Ha azonban a bejelentett típus *kisebb*, mint amit visszaadni próbálsz, akkor *kifejezett* (explicit) típusátalakítást kell alkalmaznod.

K: Kell valamit csinálnom a tagfüggvények visszaadott értékével? Vagy figyelmen kívül hagyhatom őket?

V: A Java nem követeli meg, hogy tiszteld a visszaadott értékeket. Egy tagfüggvényt meghívhatsz egy nem void típusú visszatérési értékkel úgy is, hogy a visszakapott értékkel nem foglalkozol. Ilyen esetben a tagfüggvényt azért hívod meg, mert a *benne* elvégzett munkára van szükséged, nem pedig a *visszaadott értékre*. A Javában nem kötelező felhasználnod vagy hozzárendelned valamihez a visszakapott értékeket.



Emlékeztető:
A Java odafigyel a típusokra!

Nem adhatsz vissza Zsiráfot, ha visszatérési típusként Nyulat vezettél be. Ugyanez vonatkozik a paraméterekre is: egy Nyulat váró tagfüggvénynek nem adhatsz át egy Zsiráfot.

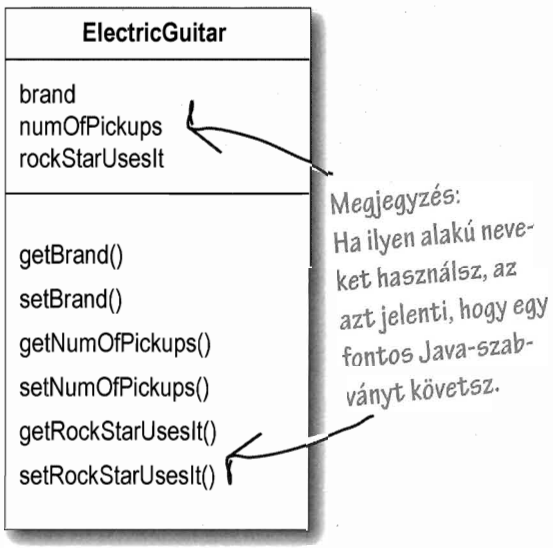
Újratöltve

- Az osztályok azt határozzák meg, hogy egy objektum mit tud és mit csinál.
- Azokat a dolgokat, amelyeket egy objektum tud, **példányváltozóknak** (állapotnak) hívjuk.
- Azokat a dolgokat, amelyeket egy objektum csinál, **tagfüggvényeknek** (viselkedésnek) nevezzük.
- A tagfüggvények példányváltozókat használnak, hogy az azonos típusú objektumok különbözőképpen viselkedhessenek.
- A tagfüggvényeknek paramétereik is lehetnek, ami azt jelenti, hogy átadhatsz nekik egy vagy több értéket.
- Az átadott értékek számának és típusának meg kell egyeznie a tagfüggvény által bevezetett paraméterek számával (és sorrendjével), illetve típusával.
- A tagfüggvényeknek átadott és onnan visszakapott értékek közvetetten előléptethetők nagyobb típusúvá, kifejezett típusátalakítással pedig kisebbé.
- A tagfüggvényeknek argumentumként átadott értékek lehetnek konkrét (literális) értékek (2, 'c', és így tovább), de a bejelentett paramétertípusnak megfelelő változók is (például *x*, ha *x* egy int típusú változó). (Argumentumként más dolgokat is átadhatsz, de még nem tartunk ott.)
- A tagfüggvényeknek *kötelező* bejelenteniük egy visszatérési értéket. A void visszatérési típus azt jelenti, hogy a tagfüggvény nem ad vissza semmit.
- Ha egy tagfüggvény nem void visszatérési típust jelent be, akkor azzal összeegyeztethető értéket *kell* visszaadnia.

Klassz dolgok, amiket a paraméterekkel és a visszatérési típusokkal csinálhatsz

Most, hogy láttad, hogyan működnek a paraméterek és a visszatérési típusok, ideje, hogy hasznos munkára fogd őket, **kiolvasóként** (getter) és **beállítóként** (setter). Ha a hivatalos stílust kedveled, *elérőknek* (accessor) és *módosítóknak* (mutator) is hívhatod őket, de ez csak időpocséklás. A „getter" és a „setter" amúgy is jobban illeszkedik a Java elnevezési rendszerébe, ezért mi maradunk a „kiolvasó" és „beállító" névnél.

A kiolvasók és beállítók segítségével – ki hitte volna? – *kiolvashatsz és beállíthatsz dolgokat* (általában példányváltozókat). Egy Kiolvasónak nincs más célja az életben, mint hogy visszatérési értékként visszaadja azt a bizonyos értéket, amit ki kell olvasnia. Bizonyára az sem ér meglepetésként, hogy a Beállító azért él, hogy fogjon egy argumentumértéket, és a felhasználásával *beállítsa* egy példányváltozó értékét.




```
class ElectricGuitar {

    String brand;
    int numOfPickups;
    boolean rockStarUsesIt;

    String getBrand() {
        return brand;
    }

    void setBrand(String aBrand) {
        brand = aBrand;
    }

    int getNumOfPickups() {
        return numOfPickups;
    }

    void setNumOfPickups(int num) {
        numOfPickups = num;
    }

    boolean getRockStarUsesIt() {
        return rockStarUsesIt;
    }

    void setRockStarUsesIt(boolean yesOrNo) {
        rockStarUsesIt = yesOrNo;
    }
}
```



Betokozás

Tedd meg, vagy azt kockáztatod, hogy megszégyenülsz vagy kinevetnek.

Eddig a kiemelkedő jelentőségű pillanatig folyamatosan elkövettük a legsúlyosabb OO-bűnök egyikét – és nem olyan apró szabálysértésekről beszélünk, mint amikor nem viszel pezsgőt a szilveszteri buliba. Nem – az egyik Fő Bűnről, nagy F-fel és nagy B-vel.

Milyen szörnyűséget követtünk el?

Felfedtük az adatainkat!

Gondtalanul fütyörészünk, miközben az adataink ott fekszenek pucéran, és *bárki* láthatja, sőt akár meg is foghatja őket. Talán már te is megtapasztaltad azt a kínos érzést, hogy milyen szem előtt hagyni a példányváltozóidat. A felfedés azt jelenti, hogy a példányváltozók elérhetővé válnak a pont művelettel, mint itt:

```
aMacska.magassag = 27;
```

```
theCat.height = 27;
```

Gondolj bele, mi történik, ha a Cat objektum méretét meghatározó példányváltozót közvetlenül módosítod egy távirányítóval: a távirányító (egy hivatkozó változó) rossz kezekbe kerülve igen veszélyes fegyver lehet. Ezt mindenképpen meg kell akadályoznod:

```
theCat.height = 0;
```

Aú! Ezt nem engedheted, hogy megtörténjen!

```
aMacska.magassag = 0;
```

Ez Szörnyű Dolog lenne. Beállító tagfüggvényeket kell írnod minden példányváltozóhoz, és módot kell találnod rá, hogy a többi kódrészt az adatok közvetlen elérése helyett a beállítók meghívására kényszerítsd.



Ha mindenkit egy beállító tagfüggvény meghívására kényszerítesz, azzal megvédheted a macskát az elfogadhatatlan méretváltoztatástól.

```
public void setHeight(int ht) {
    if (ht > 9) {
        height = ht;
    }
}
```

Betettünk egy ellenőrzést, hogy legalább minimális magasságot biztosítsunk a macskának.

Rejtsd el az adatokat!

Igen, *ennyire* egyszerű eljutni egy olyan megvalósítástól, ami szinte kiált a helytelen adatokért, egy olyanig, ami védi az adataidat *és* a jogodat, hogy később tetszés szerint módosítsd a megvalósítást.

Oké, de pontosan hogyan lehet *elrejteni* az adatokat?

A `public` és `private` elérésmódosítók segítségével.

A `public`-ot már ismered, hiszen minden main tagfüggvényben szerepelt.

Íme a betokozás (egységbe zárás, encapsulation) *első* alapszabálya (az alapszabályokra vonatkozó minden felelősségelhárítás érvényes): a példányváltozóidat ***private*** („privát") jelöléssel kell ellátnod, a hozzáférést pedig ***public*** („nyilvános") kiolvasókkal és beállítókkal célszerű szabályoznod. Ha már nagyobb tudással rendelkezel a Java nyelvű tervezésről és kódolásról, valószínűleg kicsit másképp fogod csinálni a dolgokat, de egyelőre ez a megközelítés nyújt biztonságot a számodra.

A példányváltozókat **privátként** (private) kell megjelölnöd.

A kiolvasókat és beállítókat **nyilvánosként** (public) kell bevezetned.

„Sajnos Bill elfelejtette betokozni a Macska osztályát, és a macskája kilapult."

(elcsípett beszélgetés a hűtőnél)



Java – lehull a lepel

Heti interjúnk: Egy Objektum vallomása a betokozásról

Agyhullám: Mitől olyan fontos a betokozás?

Objektum: Szokott arról álmodni, hogy 500 embernek tart előadást, és hirtelen ráébred, hogy *meztelen*?

Agyhullám: Igen, volt már ilyen álmom. Pont olyan kínos, mint az álom a Pilatesgépről, amikor…de ebbe ne menjünk bele. Rendben, tehát meztelennek érzi magát. De az alulöltözöttségen kívül van valami veszélye is a dolognak?

Objektum: Hogy van-e veszélye? Van-e *veszélye*? [hahotázni kezd] Hé, példányok, halljátok ezt? Azt kérdezi: *„Van valami veszélye is a dolognak?"* [fetreng a földön a nevetéstől]

Agyhullám: Mi ilyen vicces ebben? Szerintem ésszerű kérdés.

Objektum: Na jó, elmagyarázom. Szóval…[megint kitör belőle a nevetés, megállíthatatlanul]

Agyhullám: Kér valamit? Hozzak egy kis vizet?

Objektum: Ó, istenem. Nekem annyi. Nem, nem, jól vagyok, tényleg. Igyekszem komoly maradni. Mély levegő. Na, folytassuk.

Agyhullám: Szóval, mitől védi meg a betokozás?

Objektum: A betokozás erőteret képez a példányváltozóim körül, hogy senki ne állíthassa azokat valami – fogalmazzunk így – *illetlenre*.

Agyhullám: Tudna mondani egy példát?

Objektum: Nem kell hozzá doktori. A legtöbb példányváltozó értékét úgy kódolják, hogy bizonyos feltételezésekkel élnek az értékhatárokról. Gondoljon például azokra a dolgokra, amelyek működésképtelenné válnának, ha megengednénk a negatív számokat. Ilyen például a mosdók száma egy irodaházban, egy repülőgép sebessége, a születésnapok, egy súlyzó súlya, a mobilszámok vagy a mikrohullámú sütő energiája.

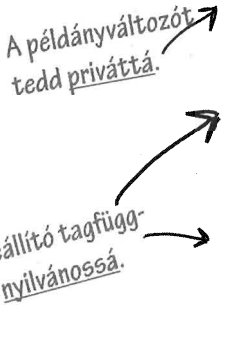
Agyhullám: Értem, mire gondol. De mivel segíti a betokozás a határok felállítását?

Objektum: Azzal, hogy a többi kódrészt arra kényszeríti, hogy beállító tagfüggvényeken keresztül működjön. Így a beállító tagfüggvény ellenőrizheti a paraméter érvényességét, és eldöntheti, hogy megfelelő-e. Előfordulhat, hogy a tagfüggvény visszautasítja a paramétert, és semmit nem csinál, de esetleg kivételt is kiválthat (mondjuk ha egy hitelkártya-kezelő alkalmazás nem létező társadalombiztosítási számot kap), vagy a legközelebbi elfogadható értékre kerekítheti a kapott paramétert. A lényeg az, hogy a beállító tagfüggvényben azt csinálhat, amit akar, míg ha a példányváltozói nyilvánosak, akkor *semmit*.

Agyhullám: De néha olyan beállító tagfüggvényekkel is találkozni, amelyek nem ellenőriznek semmit, csak beállítják az értéket. Egy olyan példányváltozó esetében, amelyet nem kell korlátozni, a beállító tagfüggvény szükségtelen többletterhelést okoz, nem? Nem lassítja le a programot?

Objektum: A beállítók (és a kiolvasók) lényege az, hogy ***később meggondolhatjuk magunkat, és nem tesszük tönkre senkinek a kódját***. Képzelje el, hogy a cég fele nyilvános példányváltozókkal használja a maga osztályát, és egy nap eszébe jut, hogy valamire nem gondolt az értékekkel kapcsolatban, ezért át kell állnia egy beállító tagfüggvény használatára. Ezzel mindenkinek a kódját tönkretenné. A betokozásban az a nagyszerű, hogy *bármikor meggondolhatja magát*, és senki sem sérül meg. A változók közvetlen használatából eredő sebességnövekedés annyira csekély, hogy ritkán éri meg a vesződséget– *vagy soha*.

A GoodDog osztály betokozása




```
class GoodDog {

  private int size;

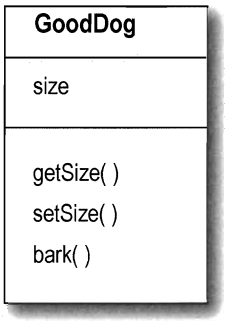
  public int getSize() {
    return size;
  }

  public void setSize(int s) {
    size = s;
  }

  void bark() {
    if (size > 60) {
      System.out.println("Wooof! Wooof!");
    } else if (size > 14) {
      System.out.println("Ruff!  Ruff!");
    } else {
      System.out.println("Yip! Yip!");
    }
  }
}

class GoodDogTestDrive {

  public static void main (String[] args) {
    GoodDog one = new GoodDog();
    one.setSize(70);
    GoodDog two = new GoodDog();
    two.setSize(8);
    System.out.println("Dog one: " + one.getSize());
    System.out.println("Dog two: " + two.getSize());
    one.bark();
    two.bark();
  }
}
```



Bár a tagfüggvények nem igazán bővítik új képességekkel a programot, az a klassz a dologban, hogy később meggondolhatod magad, visszatérhetsz, és biztonságosabbá, gyorsabbá és hatékonyabbá teheted őket.

Bárhol, ahol használhatsz egy bizonyos értéket, *használhatsz egy olyan tagfüggvényhívást is, amely ugyanolyan típusú értéket ad vissza.*

Ehelyett...

```
int x = 3 + 24;
```

...írhatod ezt is:

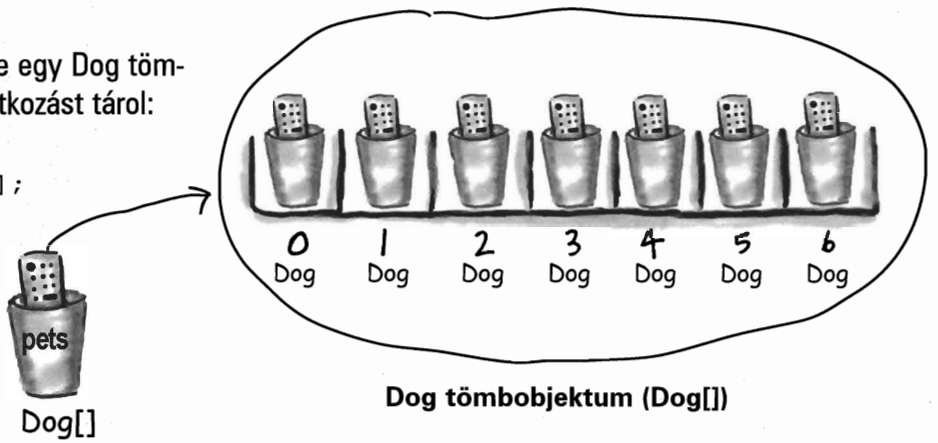
```
int x = 3 + one.getSize();
```

Hogyan viselkednek a tömbök objektumai?

Ugyanúgy, ahogy bármely más objektum. Az egyetlen különbséget az jelenti, ahogy *eléred* őket. Más szavakkal: ahogy a távirányítót kezeled. Próbáljunk meg tagfüggvényeket meghívni egy tömb Dog objektumaira!

1 Vezess be és hozz létre egy Dog tömböt, amely 7 Dog hivatkozást tárol:

```
Dog[] pets;
pets = new Dog[7];
```



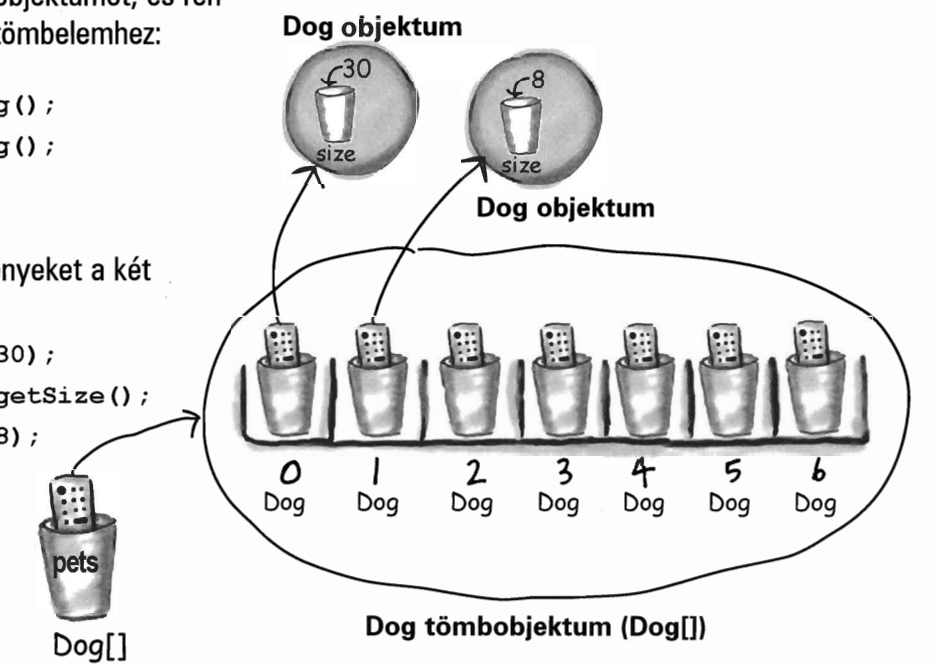
---

2 Hozz létre két új Dog objektumot, és rendeld őket az első két tömbelemhez:

```
pets[0] = new Dog();
pets[1] = new Dog();
```

3 Hívd meg a tagfüggvényeket a két Dog objektumra:

```
pets[0].setSize(30);
int x = pets[0].getSize();
pets[1].setSize(8);
```



Példányváltozók bevezetése és előkészítése

Már tudod, hogy egy változó bevezetésekor legalább egy nevet és egy típust meg kell adnod:

```
int size;
String name;
```

Azt is tudod, hogy ugyanekkor kezdőértéket is adhatsz a változónak (vagyis „előkészítheted", „inicializálhatod"):

```
int size = 420;
String name = "Donny";
```

De mi történik egy kiolvasó tagfüggvény meghívásakor, ha nem adtál kezdőértéket egy példányváltozónak? Más szavakkal, mi a példányváltozó *értéke* az előkészítés *előtt*?

A példányváltozóknak mindig van alapértelmezett értékük. Ha nem rendelsz kifejezetten értéket egy példányváltozóhoz, vagy nem hívsz meg egy beállító tagfüggvényt, a példányváltozónak akkor is van értéke.

egész változók	0
lebegőpontos változók	0.0
logikai változók	false
hivatkozások	null

```
class PoorDog {

   private int size;
   private String name;

   public int getSize() {
      return size;
   }
   public String getName() {
      return name;
   }

}

public class PoorDogTestDrive {
   public static void main (String[] args) {
      PoorDog one = new PoorDog();
      System.out.println("Dog size is " + one.getSize());
      System.out.println("Dog name is " + one.getName());
   }
}
```

bevezetünk két példányváltozót, de nem rendelünk hozzájuk értéket

Mit adnak vissza ezek?

Mit gondolsz? Egyáltalán lefordítható ez?

```
File Edit Window Help CallVet
% java PoorDogTestDrive
Dog size is 0
Dog name is null
```

A példányváltozókat nem muszáj előkészítened, mert mindig van alapértelmezett értékük. A számértékű primitíveké (a char-t is beleértve) a 0, a logikai változóké a false, az objektumhivatkozásoké pedig a null.

(Emlékezz vissza, hogy a null csupán egy olyan távirányítót jelent, ami nem vezérel semmit / nincs beprogramozva semmire. Hivatkozás, nem valódi objektum.)

Mi a különbség a példányváltozók és a helyi változók között?

1 A **példány**változókat egy osztályon belül kell bevezetni, de nem egy tagfüggvényben:

```
class Horse {
   private double height = 15.2;
   private String breed;
   // további kód...
}
```

2 A **helyi** változókat egy tagfüggvényen belül kell bevezetni:

```
class AddThing {
   int a;
   int b = 12;

   public int add() {
      int total = a + b;
      return total;
   }
}
```

3 A **helyi** változóknak KÖTELEZŐ kezdőértéket adni használat előtt:

```
class Foo {
   public void go() {
      int x;
      int z = x + 3;
   }
}
```

Nem fordítható le!! Az x-et érték nélkül is bevezetheted, de amint megpróbálod HASZNÁLNI, a fordítóprogram bepánikol.

```
File Edit Window Help Yikes
% javac Foo.java
Foo.java:4: variable x might
not have been initialized
      int z = x + 3;
1 error        ^
```

A helyi változók NEM kapnak alapértelmezett értéket! A fordító panaszkodik, ha egy helyi változót az előtt próbálsz használni, hogy értéket adtál volna neki.

Nincsenek

hülye kérdések

K: Mi a helyzet a tagfüggvény-paraméterekkel? Rájuk is érvényesek a helyi változókra vonatkozó szabályok?

V: A tagfüggvény-paraméterek lényegében ugyanolyanok, mint a helyi változók – a tagfüggvényen *belül* vezetjük be őket (pontosabban a tagfüggvény *argumentumlistájában*, nem pedig a tagfüggvény *törzsében*, de attól még helyi, és nem példányváltozók). Ugyanakkor a tagfüggvény-paraméterek előkészítése soha nem vonható vissza, így nem is kapsz olyan hibaüzenetet a fordítótól, hogy egy paraméterváltozó nincs előkészítve.

Ennek az az oka, hogy a fordító hibát jelez, ha anélkül próbálsz meghívni egy tagfüggvényt, hogy átadnád neki azokat az argumentumokat, amelyekre szüksége van.

A paraméterek tehát MINDIG rendelkeznek értékkel, mert a fordító garantálja, hogy a tagfüggvények meghívása mindig olyan argumentumokkal történjen, amelyek megfelelnek a tagfüggvény bevezetésekor bejelentett paramétereknek, és az argumentumok automatikusan hozzárendelődnek a paraméterekhez.

Változók (primitívek vagy hivatkozások) összehasonlítása

Előfordulhat, hogy tudni szeretnéd, hogy két *primitív* azonos-e. Ezt elég könnyű kideríteni – csak az == műveletet kell használnod. Néha viszont azt szeretnéd tudni, hogy két hivatkozó változó egy objektumra hivatkozik-e a kupacban. Ez sem nehéz – ismét csak az == művelethez fordulhatsz. Az is lehet azonban, hogy két *objektumról* szeretnéd kideríteni, hogy egyenértékűek-e. Ehhez már az `.equals()` tagfüggvény szükséges. Az objektumok egyenlősége az objektum típusától függ. Például ha két különböző String objektum ugyanazokból a karakterekből áll, akkor értelmileg egyenértékűek, függetlenül attól, hogy a kupac két különböző objektumáról van-e szó. De mi a helyzet egy Kutyával? Két Kutya objektum azonos, ha történetesen azonos a méretük és a súlyuk? Valószínűleg nem. Az tehát, hogy két objektumot egyenlőként kell-e kezelni, attól függ, hogy minek van értelme az adott objektumtípus esetében. Az objektumegyenlőség elvére a későbbi fejezetekben (és a B függelékben) még visszatérünk; egyelőre elég annyit tudnod, hogy az == műveletet *kizárólag* két változó bitjeinek az összehasonlítására használjuk. Az, hogy *mit* ábrázolnak ezek a bitek, nem számít. A bitek vagy megegyeznek, vagy nem.

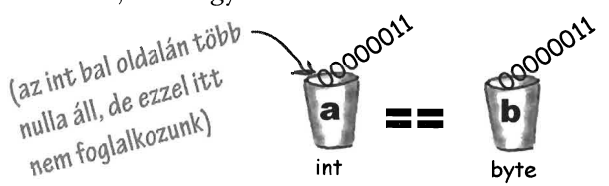
Ha két primitívet szeretnél összehasonlítani, vagy azt szeretnéd tudni, hogy két hivatkozás *ugyanarra* az objektumra mutat-e, használd az == műveletet.
Ha azt szeretnéd tudni, hogy két *különböző* objektum egyenlő-e, használd az equals() tagfüggvényt.

(Például ha azt akarod kideríteni, hogy két különböző String objektum egyaránt a „Fred" karaktereket ábrázolja-e.)

Két primitív összehasonlításához használd az == műveletet

Az == művelet bármilyen fajta két változó összehasonlítására használható, és semmi mást nem tesz, mint hogy összehasonlítja a biteket.
Az if (a == b) [...] utasítás megvizsgálja az a és a b bitjeit, és true-t ad vissza, ha a bitminta megegyezik (bár a változó méretével nem törődik, ezért figyelmen kívül hagyja a bal szélen álló nullákat).

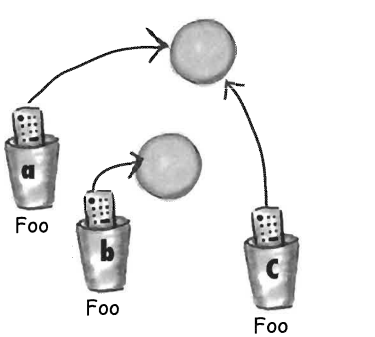
```
int a = 3;
byte b = 3;
if (a == b) { // true }
```



Ha azt szeretnéd tudni, hogy két hivatkozás egyenlő-e (ami azt jelenti, hogy ugyanarra az objektumra hivatkoznak a kupacban), használd az == műveletet

Ne feledd, az == művelet csak a változóban található bitmintát nézi. A szabályok ugyanazok, akár hivatkozásról, akár primitív változóról van szó. Az == művelet tehát true értéket ad vissza akkor is, ha két hivatkozó változó ugyanarra az objektumra hivatkozik! Ebben az esetben nem ismered a bitmintát (mert az a JVM-től függ, és rejtve marad előtted), azt viszont *tudod*, hogy bárhogy nézzen is ki, *ugyanaz lesz, ha a két hivatkozás egyetlen objektumra mutat.*

```
Foo a = new Foo();
Foo b = new Foo();
Foo c = a;
if (a == b) { // false }
if (a == c) { // true }
if (b == c) { // false }
```



az a == c igaz (true)
az a == b hamis (false)



Ragaszd fel!



Vörös a rózsa,
darabos e vers,
adj át érték szerint,
s másolatot nyersz.

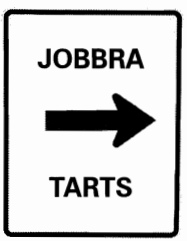
Miért, te jobb verset tudsz írni? Próbáld ki! Cseréld le a versünk ostoba második sorát a sajátodra – de még jobb, ha az egészet átírod, hogy tényleg *soha* ne felejtsd el.

Hegyezd ki a ceruzád!

Melyik érvényes?

Az alábbi tagfüggvényt alapul véve mely tagfüggvényhívások érvényesek a jobboldalt felsoroltak közül?

Pipáld ki azokat a hívásokat, amelyek érvényesnek minősülnek! (Egyes utasítások a tagfüggvényhívásokban használt értékek hozzárendelésére szolgálnak.)




```
int calcArea(int height, int width) {
   return height * width;
}
```

```
int a = calcArea(7, 12);
short c = 7;
calcArea(c,15);
int d = calcArea(57);
calcArea(2,3);
long t = 42;
int f = calcArea(t,17);
int g = calcArea();
calcArea();
byte h = calcArea(4,20);
int j = calcArea(2,3,5);
```



Gyakorlat



Légy a fordító!

Az ezen az oldalon található Java-fájlok mindegyike egy-egy teljes forrásfájl. A te feladatod az, hogy eljátszd a fordítóprogram szerepét, és kiderítsd, hogy lefordíthatók-e ezek a fájlok. Ha nem, hogyan javítanád ki őket? Ha igen, milyen kimenetet eredményeznek?

A

```
class XCopy {

  public static void main(String [] args) {

    int orig = 42;

    XCopy x = new XCopy();

    int y = x.go(orig);

    System.out.println(orig + " " + y);
  }

  int go(int arg) {

    arg = arg * 2;

    return arg;
  }
}
```

B

```
class Clock {
  String time;

  void setTime(String t) {
    time = t;
  }

  void getTime() {
    return time;
  }
}

class ClockTestDrive {
  public static void main(String [] args) {

    Clock c = new Clock();

    c.setTime("1245");
    String tod = c.getTime();
    System.out.println("time: " + tod);
  }
}
```



Gyakorlat

Néhány Java-összetevő kosztümbe öltözve álarcosbált tart. Adnak neked egy kis segítséget, és ki kell találnod, hogy ki kicsoda. Tegyük fel, hogy mindig igazat állítanak magukról. Ha olyasmit találnak mondani, ami többükre is jellemző lehet, válaszd ki mindegyiket, akire ráillik a leírás, és írd a nevüket az adott állítás mellé.

A mai vendégek:
példányváltozó, argumentum, visszatérési érték, kiolvasó, beállító, betokozás, public, private, érték szerinti átadás, tagfüggvény



Egy osztályban bármennyi lehet belőlem. _____

Egy tagfüggvényben csak egy lehet belőlem. _____

Közvetetten elő lehet léptetni. _____

A privát példányváltozókat részesítem előnyben. _____

Valójában az a jelentésem, hogy „másolat készítése". _____

Csak a kiolvasóknak lenne szabad frissíteniük. _____

Egy tagfüggvényben több is lehet belőlem. _____

Mindenképpen visszaadok valamit. _____

Engem nem szabad példányváltozókkal használni. _____

Több argumentumom is lehet. _____

Csak egy argumentumot fogadok el. _____

Segítem a betokozást. _____

Én mindig egyedül járok. _____



Összekevert ÜZENETEK

Nézd meg a jobb oldalon található rövid Java-programot. A programból két kódblokk hiányzik. A feladatod az, hogy **összepárosítsd az alábbi kódjelölteket** azzal a **kimenettel**, amit akkor látnál, ha az adott blokkokat szúrnád be a kódba. Nem minden kimenetsornak van párja, viszont egyes kimenetsorok többször is felhasználhatók. Kösd össze vonalakkal a kódjelölteket és a nekik megfelelő parancssori kimeneteket!

```
public class Mix4 {
  int counter = 0;
  public static void main(String [] args) {
    int count = 0;
    Mix4 [] m4a  =new Mix4[ 20] ;
    int x = 0;
    while ( [          ] ) {
      m4a[ x]  = new Mix4();
      m4a[ x] .counter = m4a[ x] .counter + 1;
      count = count + 1;
      count = count + m4a[ x] .maybeNew(x);
      x = x + 1;
    }
    System.out.println(count + " "
                       + m4a[ 1] .counter);
  }

  public int maybeNew(int index) {
    if ( [          ] ) {
      Mix4 m4 = new Mix4();
      m4.counter = m4.counter + 1;
      return 1;
    }
    return 0;
  }
}
```

Kódjelöltek:

```
x < 9
index < 5
```

```
x < 20
index < 5
```

```
x < 7
index < 7
```

```
x < 19
index < 1
```

Lehetséges kimenetek:

```
14 7
```

```
9 5
```

```
19 1
```

```
14 1
```

```
25 1
```

```
7 7
```

```
20 1
```

```
20 5
```



Halászd ki a medencéből!



A ***feladatod*** az, hogy kihalászd a kóddarabokat a medencéből, és elhelyezd azokat a kód üresen hagyott helyein. Egy kóddarabot **nem** használhatsz fel egynél többször, viszont nem lesz szükséged minden darabra.

A ***cél*** egy olyan osztály felépítése, amely lefordítható, futtatható, és az alább látható kimenetet eredményezi.

Kimenet:

```
File  Edit  Window  Help  BellyFlop
%java Puzzle4
result 543345
```

Megjegyzés: A medencében úszó kóddarabok mindegyikét csak egyszer használhatod fel!

```
public class Puzzle4 {
  public static void main(String [] args) {
    _____________________________________
    int y = 1;
    int x = 0;
    int result = 0;
    while (x < 6) {
      _____________________________
      _____________________________
      y = y * 10;
      ________________
    }
    x = 6;
    while (x > 0) {
      ________________
      result = result + ____________________
    }
    System.out.println("result " + result);
  }
}
class ___________ {
  int ivar;
  ________ ______ doStuff(int _________) {
    if (ivar > 100) {
      return ____________________________
    } else {
      return ____________________________
    }
  }
}
```

```
doStuff(x);
obs.doStuff(x);
obs[x].doStuff(factor);
obs[x].doStuff(x);
ivar = x;
obs.ivar = x;
obs[x].ivar = x;
obs[x].ivar = y;
ivar
factor
public
private
ivar + factor;
ivar * (2 + factor);
ivar * (5 - factor);
ivar * factor;
x = x + 1;
x = x - 1;
Puzzle4
Puzzle4b
Puzzle4b( )
int
short
Puzzle4 [ ] obs = new Puzzle4[6];
Puzzle4b [ ] obs = new Puzzle4b[6];
Puzzle4b [ ] obs = new Puzzle4[6];
obs [x] = new Puzzle4b(x);
obs [ ] = new Puzzle4b( );
obs [x] = new Puzzle4b( );
obs = new Puzzle4b( );
```



Kutyaszorítóban Stim-City-ben

Amikor Buchanan Jai oldalába nyomta a sokkolóját, Jai mozdulatlanná dermedt. Tudta, hogy Buchanan éppoly ostoba, mint amilyen ocsmány, ezért nem akart ujjat húzni a behemóttal. Buchanan a főnöke irodájába parancsolta Jait. Mivel semmi rosszat nem tett (mostanában), arra a következtetésre jutott, hogy egy kis csevely Levelerrel, Buchanan főnökével, nem lehet olyan vészes. A nyugati parton sok-sok neurostimulátort eladott az utóbbi időben, aminek – úgy vélte – Leveler biztosan örülni fog. A feketepiaci stimulátorok nem jelentettek éppen aranybányát, de meglehetősen ártalmatlanok voltak. A legtöbb stimfüggő, akit ismert, leállt egy idő után, és visszatért a régi életéhez, és talán csak egy kicsit volt zizibb, mint azelőtt.

Leveler „irodája" egy lepukkantnak tűnő hajó volt, de miután Buchanan belökte az ajtón, Jai látta, hogy alaposan felturbózták, hogy kellően gyors és felfegyverzett legyen egy olyan helyi nagykutyának, mint Leveler. „Jai, fiam" – sziszegte Leveler – „jó viszontlátni téged". „Én is örülök..." – válaszolta Jai, érzékelve a gúnyt Leveler hangjában – „Ha jól emlékszem, mi rendben vagyunk, nem? Vagy lemaradtam valamiről?" „Hahaha! Jól adod, Jai...csakhogy nemrég valaki – hogy is mondjam? – megsértette a köreimet."

Jai önkéntelenül megrándult. Fénykorában az egyik legmenőbb hackernek tartották – talán ezért is volt, hogy amint valaki rájött, hogyan törje fel egy nagymenő biztonsági rendszerét, Jai rögtön a nemkívánatos figyelem középpontjába került. „Az tuti, hogy nem én voltam, haver!" – tiltakozott – „Nekem nem éri meg. Visszavonultam. Csak mozgatom az árumat, és nem ütöm bele az orrom mások dolgába." „Persze, persze" – nevetett Leveler – „Biztos vagyok benne, hogy ehhez semmi közöd, de nagy pénzt vesztek, amíg le nem kapcsoljuk ezt az új hackerbajnokot." „Hát, sok szerencsét hozzá, Leveler. Akkor itt ki is tehetnél, és hozok néhány 'egységet' neked, mielőtt bezárom a boltot." – próbálkozott Jai.

Ötperces rejtély



„Attól tartok, ez nem lesz ilyen egyszerű, Jai. Buchanan szerint az a hír járja, hogy otthon vagy a J37NE-ben." – tapogatózott Leveler. „Neural Edition? Persze, volt dolgom vele, és akkor mi van?" – kérdezte óvatosan Jai. „Azzal tudatom a stimmerekkel, hogy hová érkezik a következő szállítmány" – magyarázta Leveler – „Csak az a gond, hogy valamelyikük elég ideig maradt tiszta ahhoz, hogy kitalálja, hogyan törhet be a WareHousing adatbázisomba. Kell valaki, akinek olyan gyorsan jár az agya, mint neked, hogy vessen egy pillantást a StimDrop J37NE osztályomra – a tagfüggvényekre, a példányváltozókra, az egész *enchiladára* –, és kiderítse, hol jutottak be. Talán a..." „Hé!" – tiltakozott hevesen Buchanan – „Az én kódomat nem fogja fürkészni egy olyan mocskos hacker, mint Jai!" „Nyugi, haver" – látta meg a lehetőséget Jai – „Biztos vagyok benne, hogy király munkát végeztél az elérésmódosí..." „Naná, bittologató!" – vicsorgott Buchanan – „Minden junkie-szintű tagfüggvényt nyilvánossá tettem, hogy a stimmerek hozzáférjenek a szállítási hely adataihoz, de az összes létfontosságú WareHousing-tagfüggvényt privátként jelöltem meg. Senki sem érheti el azokat kívülről, haver, senki!"

„Azt hiszem, meg tudom találni a rést, Leveler. Mit szólnál, ha itt a sarkon kitennénk Buchanant, és mennénk egy kört?" – javasolta Jai. Buchanan felhördült, Jainek szegezte a sokkolóját, de Leveler fegyvere már ott volt a nyakán. „Nyugi, Buchanan." – vigyorgott Leveler – „Dobd el a sokkolót, és húzz el szépen sétálni. Jainek és nekem megbeszélnivalónk van."

Mire gyanakodott Jai?
Ép bőrrel fogja elhagyni Leveler hajóját?



Gyakorlat – Megoldás

B

```
class Clock {
  String time;
  void setTime(String t) {
    time = t;
  }
  String getTime() {
    return time;
  }
}

class ClockTestDrive {
  public static void main(String [] args) {
    Clock c = new Clock();
    c.setTime("1245");
    String tod = c.getTime();
    System.out.println("time: " + tod);
  }
}
```

Megjegyzés: a kiolvasó tagfüggvények értelemszerűen rendelkeznek visszatérési értékkel.

A Az „XCopy" osztály így, ahogy van, lefordítható és futtatható. A kimenete ez lesz: „42 84". Ne feledd, hogy a Java érték szerinti átadást alkalmaz (ami másolásos átadást jelent), tehát az „orig" változót a go() tagfüggvény nem változtatja meg.

Egy osztályban bármennyi lehet belőlem.	**példányváltozó, kiolvasó, beállító, tagfüggvény**
Egy tagfüggvényben csak egy lehet belőlem.	**visszatérési érték**
Közvetetten elő lehet léptetni.	**visszatérési érték, argumentum**
A privát példányváltozókat részesítem előnyben.	**betokozás**
Valójában az a jelentésem, hogy „másolat készítése".	**érték szerinti átadás**
Csak a kiolvasóknak lenne szabad frissíteniük.	**példányváltozó**
Egy tagfüggvényben több is lehet belőlem.	**argumentum**
Mindenképpen visszaadok valamit.	**kiolvasó**
Engem nem szabad példányváltozókkal használni.	**public**
Több argumentumom is lehet.	**tagfüggvény**
Csak egy argumentumot fogadok el.	**beállító**
Segítem a betokozást.	**kiolvasó, beállító, public, private**
Én mindig egyedül járok.	**visszatérési érték**

Fejtörők – Megoldás

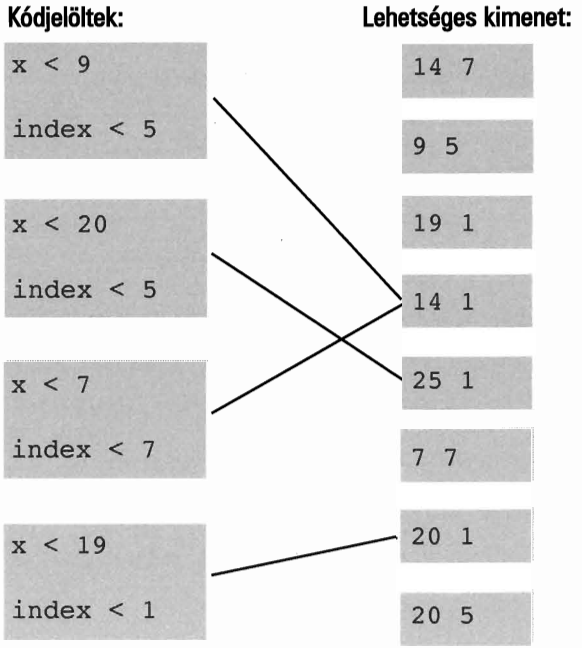
```
public class Puzzle4 {
  public static void main(String [] args) {
    Puzzle4b [ ] obs = new Puzzle4b[6];
    int y = 1;
    int x = 0;
    int result = 0;
    while (x < 6) {
      obs[x] = new Puzzle4b( );
      obs[x] . ivar = y;
      y = y * 10;
      x = x + 1;
    }
    x = 6;
    while (x > 0) {
      x = x - 1;
      result = result + obs[x].doStuff(x);
    }
    System.out.println("result " + result);
  }
}
class Puzzle4b {
  int ivar;
  public int doStuff(int factor) {
    if (ivar > 100) {
      return ivar * factor;
    } else {
      return ivar * (5 - factor);
    }
  }
}
```

Kimenet:

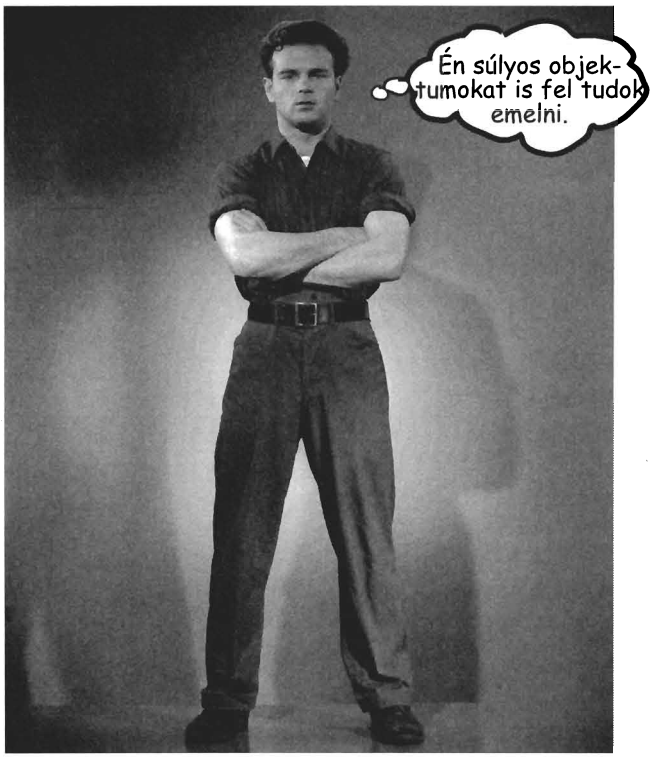
```
File Edit Window Help BellyFlop
%java Puzzle4
result 543345
```

Az „Ötperces rejtély" megfejtése:

Jai tudta, hogy Buchanannek nem vág éppen élesen az esze. Amikor Buchanan a kódjáról beszélt, nem említette a példányváltozókat. Jai gyanította, hogy bár Buchanan helyesen kezelte a tagfüggvényeit, a példányváltozóit elfelejtette privátként megjelölni. Ez a banánhéj könnyen kerülhetett pár ezresébe Levelernek.



Bivalyerős tagfüggvények



Pumpáld fel a tagfüggvényeid izmait! Eddig változókkal és néhány objektummal játszadoztál, valamint írtál némi kódot. De még nem vagy elég erős. Több eszközre van szükséged. Például **műveletekre**: olyan műveletekre, amelyekkel érdekesebb dolgokat csinálhatsz, mint az *ugatás*. És **ciklusokra**. Nem olyan bágyadtakra, mint a *while* – ha komolyan veszed a dolgot, akkor ***for*** ciklusok kellenek. Hasznos lenne **véletlenszámokat is előállítani**. Vagy egy **String-et int-té alakítani** – ó, az igazán klassz lenne! Muszáj ezt is megtanulnod. És miért ne tanulnád meg úgy az egészet, hogy *építesz* valami igazit, hogy lásd, milyen a semmiből megírni (és leteszteltni) egy programot. Mondjuk egy olyan **játékot**, mint a Torpedó. Ez már komoly súlyemelés, ezért *két* fejezet kell majd a feladat befejezéséhez. Ebben a fejezetben egy egyszerű változatot fogsz készíteni, aztán a 6. fejezetben építesz egy nagyobb tudású, deluxe kiadást.

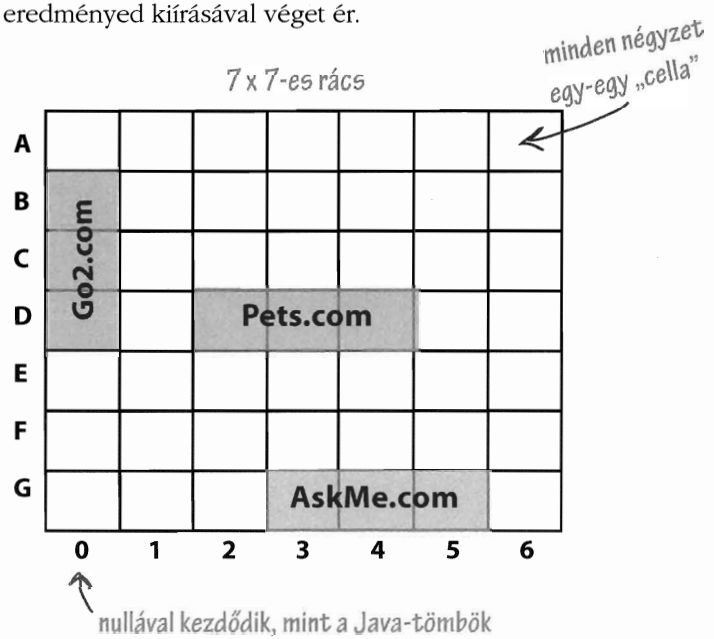
Építs torpedójátékot – Sink a Dot Com

A játékban a számítógép ellen küzdesz, de az eredeti Torpedóval szemben ebben a játékban nem helyezel el saját hajókat. A feladatod ehelyett az, hogy a lehető legkevesebb lövéssel elsüllyeszd a számítógép hajóit.
Ja, nem is hajókat kell elsüllyesztened, hanem Dot Com cégeket! (Csak hogy üzleti szempontból meg tudd indokolni a könyvre költött pénzt.)

A cél: Elsüllyeszteni a számítógép Dot Com-jait, a lehető legkevesebb lövéssel. A végén a teljesítményed alapján értékelést vagy szintbesorolást kapsz.

Előkészítés: Amikor elindítod a programot, a számítógép három Dot Com-ot helyez el egy 7 x 7-es virtuális rácson. Ha ezzel elkészült, a játék kéri az első tippet.

A játék menete: Még nem tanultad meg, hogyan kell grafikus felületet építeni, ezért ez a változat parancssori utasításokkal működik. A számítógép felszólít, hogy írj be egy tippet (egy cella koordinátáit), amit így adhatsz meg: „A3", „C5" stb. A lövés eredményét is a parancssorban láthatod, „Hit" (Talált), „Miss" (Nem talált) vagy „You sunk Pets.com" (Pets.com talált, süllyedt) formában (vagy bármelyik Dot Com cégnek legyen is szerencsés napja). Ha mind a három Dot Com-ot a 404-es süllyesztőbe küldted, a játék az elért eredményed kiírásával véget ér.



A Sink a Dot Com játékot fogod elkészíteni, egy 7 x 7-es ráccsal és három Dot Com céggel. Minden Dot Com három cellát foglal el.

részlet a játék üzeneteiből

```
File Edit Window Help Sell
%java DotComBust
Enter a guess  A3
miss
Enter a guess  B2
miss
Enter a guess  C4
miss
Enter a guess  D2
hit
Enter a guess  D3
hit
Enter a guess  D4
Ouch! You sunk Pets.com   : (
kill
Enter a guess  B4
miss
Enter a guess  G3
hit
Enter a guess  G4
hit
Enter a guess  G5
Ouch! You sunk AskMe.com   : (
```

Első lépés: a magasszintű terv elkészítése

Azt már tudod, hogy osztályokra és tagfüggvényekre lesz szükséged, csak az a kérdés, hogy milyenekre. Ahhoz, hogy választ tudj adni erre a kérdésre, több információ kell a játék feladatairól. Először is, meg kell határoznod a játék általános menetét. Íme a vázlat:

1 A felhasználó elindítja a játékot

- **A** A játék létrehoz három Dot Com-ot
- **B** A játék elhelyezi a három Dot Com-ot egy virtuális rácson.

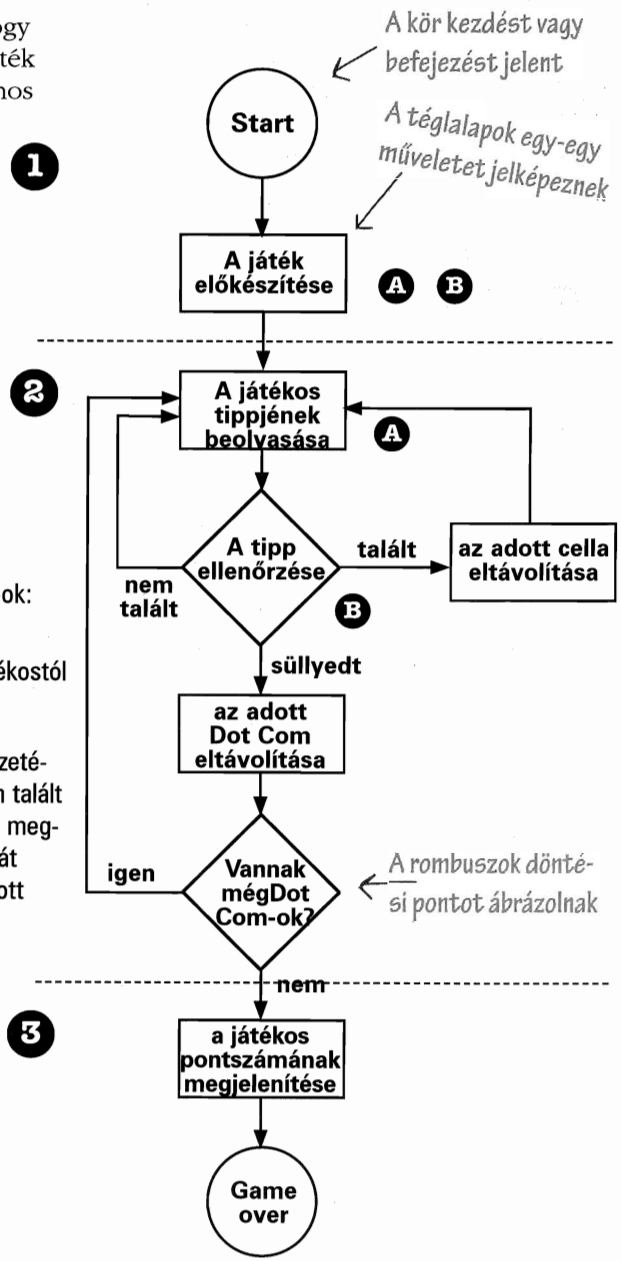
2 A játék megkezdődik

Az alábbiak addig ismétlődnek, amíg el nem fogynak a Dot Com-ok:

- **A** A program kér egy tippet („A2", „C0" stb.) a játékostól
- **B** A program összeveti a tippet a Dot Com-ok helyzetével, hogy meghatározza, hogy a lövés talált, nem talált vagy elsüllyesztett-e valamit, majd végrehajtja a megfelelő műveletet: találat esetén törli az adott cellát (A2, D4 stb.), „süllyedt" esetén pedig törli az adott Dot Com-ot.

3 A játék befejeződik

A program értékeli a játékost a lövések száma alapján.



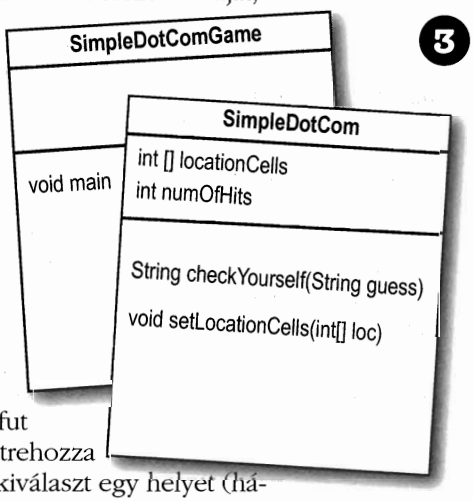
Fúú! Egy igazi folyamatábra!

Most már képben vagy azt illetően, hogy milyen dolgokat kell csinálnia a programnak. A következő lépés annak kiderítése, hogy milyen **objektumokra** van szükséged a feladatok elvégzéséhez. Ne feledd: gondolkodj úgy, mint Brad, ne úgy, mint Larry – előbb ne az ***eljárásokra***, hanem a programban található ***dolgokra*** összpontosíts.

Az „Egyszerű Dot Com Játék” – kíméletesebb bevezetés

Úgy néz ki, legalább két osztályra lesz szükséged: egy Game (Játék) és egy DotCom nevűre. Mielőtt azonban felépítenéd a teljes fegyverzetű ***Sink a Dot Com*** játékot, először egy lecsupaszított, leegyszerűsített változatot fogsz készíteni belőle. Legyen ennek a neve ***Simple Dot Com Game***, vagyis „Egyszerű Dot Com Játék”. Az egyszerű változat még *ebben* a fejezetben elkészül, hogy aztán a *következő* fejezetben nekiláthass a deluxe kiadásnak. A játék első változatában minden egyszerűbb lesz. Kétdimenziós rács helyett a Dot Com-hajókat egyetlen *sorban* fogjuk elrejteni, és nem is *három*, hanem csak *egy* Dot Com-ra fogsz lőni.

A játék célja persze ugyanaz marad, tehát ugyanúgy létre kell hozni egy DotCom-példányt, hozzá kell rendelni egy helyhez valahol a sorban, ki kell olvasni a felhasználó tippjét, és ha sikerült eltalálni a DotCom összes celláját, akkor be kell fejezni a játékot. Az egyszerűsített változat remek kiindulópont lesz a teljes játék felépítéséhez: ha működőképessé tudod tenni ezt a változatot, akkor az összetettebb változat elkészítése már gyerekjáték lesz.



Az első változatban a Game osztálynak nem lesznek példányváltozói, és a játék teljes kódja a main() tagfüggvényben kap helyet. Más szavakkal, amikor a program elindul, és lefut a main() tagfüggvény, a játék létrehozza az egyetlen DotCom-példányt, kiválaszt egy helyet (három egymást követő cellát az egyetlen virtuális, hét cellából álló sorban) a számára, kér egy tippet a felhasználótól, megnézi, hogy talált-e a lövés, és ezt addig ismétli, amíg el nem sikerül találni mind a három cellát.

Ne feledd, hogy a virtuális sor…*virtuális*, vagyis konkrétan sehol nem létezik a programban. Amíg a játék és a játékos egyaránt tudja, hogy a DotCom három egymást követő cellában rejtőzik a lehetséges hétből (a számozás nullától indul), magát a sort nem kell ábrázolni a kódban. Csábítást érezhetsz, hogy létrehozz egy hét int értékből álló tömböt, majd a DotCom-ot hozzárendeld a tömb három eleméhez, de ez nem szükséges. Csupán egy olyan tömbre van szükséged, amely a DotCom által elfoglalt három cellát tárolja.

1 **A program elindul,** létrehoz EGY DotCom-ot, és hozzárendel három cellát az egyetlen, hét cellából álló sorból.

Olyan koordináták helyett, mint az „A2”, „C4” stb., a helyzetet egyszerű egész értékek adják meg (az alábbi ábrán például a DotCom celláit az 1,2,3 jelöli).



2 **Megkezdődik a játék.** Kérünk egy tippet a játékostól, és megnézzük, hogy eltalálta-e a DotCom három cellájának valamelyikét. Ha igen, növeljük a numOfHits (találatok száma) változó értékét.

3 **A játék befejeződik,** ha mindhárom cellát találat érte (vagyis a numOfHits változó értéke 3). A program ekkor kiírja a felhasználónak, hogy hány tipp kellett a DotCom elsüllyesztéséhez.

Egy teljes játék

```
File Edit Window Help Destroy
%java SimpleDotComGame
enter a number  2
hit
enter a number  3
hit
enter a number  4
miss
enter a number  1
kill
You took 4 guesses
```

Hogyan építhetsz fel egy osztályt?

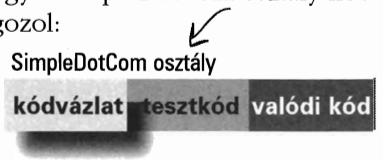
Programozóként valószínűleg van valamilyen kedvenc módszered, eljárásod, illetve megközelítésed, amit követsz, amikor kódot írsz. Nos, nekünk is. Az alább bemutatott eljárás lépéseit úgy alakítottuk ki, hogy segítsenek neked megérteni (és megtanulni), hogy mit kell végiggondolnod, amikor megírod egy osztály kódját. A valóságban persze nem feltétlenül ezeket a lépéseket követjük (ez *rád* is áll), hiszen az alkalmazott megközelítést ott már befolyásolják az adott projekt követelményei, a személyes ízlés, illetve a munkaadó előírásai. Mi viszont abban a szerencsés helyzetben vagyunk, hogy lényegében azt csinálunk, amit akarunk. Amikor pedig „szemléltetés céljából" hozunk létre egy Java-osztályt, általában így csináljuk:

- Meghatározzuk, mit kell *csinálnia* az osztálynak.
- Listát készítünk a **példányváltozókról és tagfüggvényekről**.
- **Kódvázlatot** írunk a tagfüggvények számára (erre mindjárt visszatérünk).
- **Tesztkódot** írunk a tagfüggvények számára.
- **Megvalósítjuk** („implementáljuk") az osztályt.
- **Teszteljük** a tagfüggvényeket.
- Szükség szerint **hibamentesítjük** a kódot, és **új megvalósítást** írunk.
- Hálát adunk, amiért az úgynevezett *„szemléltető"* alkalmazásunkat nem valódi, élő felhasználókon kell kipróbálnunk.

A három dolog, amit minden osztály esetében megírunk:



A következő néhány oldal fejlécében ez a szürke sáv mutatja majd, hogy hol tartasz. Például ha az alábbi ábrát látod a lap tetején, az azt jelenti, hogy a SimpleDotCom osztály kódvázlatán dolgozol:



kódvázlat

Egyfajta álkód (pseudocode, pszeudokód), ami segít, hogy a logikára összpontosíthass, és ne kelljen a nyelvtan miatt stresszelned.

tesztkód

Osztály vagy tagfüggvények, amelyek ellenőrzik a valódi kódot, és meggyőződnek róla, hogy helyesen működik-e.

valódi kód

Az osztály tényleges megvalósítása. Ez az, amit úgy hívunk, hogy valódi Java-kód.

Feladatok:

SimpleDotCom osztály

- kódvázlatot írni
- tesztkódot írni
- megírni a végleges Java-kódot

SimpleDotComGame osztály

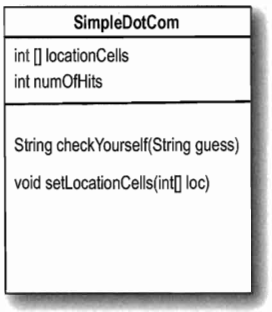
- kódvázlatot írni
- tesztkódot írni [nem]
- megírni a végleges Java-kódot



Agytorna

Dolgoztasd meg azokat az agysejteket!

Mi alapján döntenéd el, hogy melyik osztályt vagy mely osztályokat kell először felépíteni, amikor programot írsz? Feltételezve, hogy a legapróbbaktól eltekintve minden program egynél több osztályból áll (már ha követed a helyes OO-elveket, és egy osztállyal *nem* végeztetsz el többféle feladatot), hol kezdenéd?



Ebből a példából láthatod majd, hogyan működik a kódvázlat (a mi álkódunk): valahol félúton áll a valódi Java-kód és az osztály hétköznapi nyelvű leírása között. A legtöbb kódvázlat három részre oszlik: a példányváltozók bevezetésére, a tagfüggvények bevezetésére, és a tagfüggvények logikájára. A vázlat legfontosabb része a tagfüggvénylogika, mert ez határozza meg, *mi* történjen, amit aztán lefordítunk arra, hogy *hogyan*, amikor ténylegesen megírjuk a tagfüggvény kódját.

BEVEZET ***int tömb*** a helyzetcellák tárolására. A neve legyen ***locationCells***.
BEVEZET ***int*** a találatok számának tárolására. A neve legyen ***numOfHits***, és **BEÁLLÍT** 0-ra.

BEVEZET ***checkYourself()*** tagfüggvény, amely egy ***String***-et vár a felhasználó tippjeként (userGuess – „1", „3" stb.), ellenőrzi, és visszaad egy eredményt, ami jelzi, hogy „talált" (hit), „nem talált" (miss) vagy „süllyedt" (kill).
BEVEZET ***setLocationCells()*** beállító tagfüggvény, amely egy ***int tömb***öt vár (amelyben a három cellahelyzet található int-ként – 2,3,4 stb.)

TAGFÜGGVÉNY: ***String checkYourself(String userGuess)***

KIOLVAS a felhasználó tippje String paraméterként
ÁTALAKÍT a felhasználó tippje ***int***-té
ISMÉTEL az ***int*** tömb minden helyzetcellájával
// ÖSSZEVET a felhasználó tippje a helyzetcellával
IF a felhasználó tippje egyezik
NÖVEL találatok száma
// KIDERÍT hogy az utolsó helyzetcella volt-e
IF találatok száma 3, **RETURN** „kill" mint eredmény
ELSE nem süllyedt, ezért **RETURN** „hit"
IF VÉGE
ELSE a felhasználó tippje nem egyezik, ezért **RETURN** „miss"
IF VÉGE
ISMÉTEL VÉGE
TAGFÜGGVÉNY VÉGE

TAGFÜGGVÉNY: ***void setLocationCells(int[] cellLocations)***

KIOLVAS cellahelyzetek ***int tömb*** paraméterként
HOZZÁRENDEL cellahelyzet-paraméter a cellahelyzetek példányváltozójához
TAGFÜGGVÉNY VÉGE

kódvázlat **tesztkód** valódi kód

A tagfüggvények megvalósítása



Írd meg a valódi tagfüggvénykódokat most, és hozd működésbe a drágaságot!

Mielőtt azonban nekilátnál a tagfüggvények kódba öntésének, lépj hátra egyet, és írj egy kis kódot a tagfüggvények *tesztelésére*. Igen, a tesztkódot *előbb* írod meg, mint hogy bármi is lenne, amit tesztelhetnél!

A tesztkód megírásának elsőbbsége az extrém programozás (XP) egyik fogása, amely megkönnyítheti (és felgyorsíthatja) a kód megírását. Nem azt mondjuk, hogy feltétlenül az extrém programozás elveit kell követned, de mi is szeretjük először a teszteket megírni. Ráadásul olyan *királyul* hangzik, hogy „XP"!

Extrém programozás (XP)

Az extrém programozás (Extreme Programming, XP) újonc a szoftverfejlesztési módszertanok között. Az XP, amelyről sokan úgy vélik, hogy „az a mód, ahogy a programozók valóban dolgozni akarnak", az 1990-es évek végén jelent meg, és azóta számos cég tette magáévá az elveit, a kétszemélyes garázsműhelyektől a Ford Motor Company-ig. Az XP alaptétele, hogy az ügyfél mindig azt kapja, amit akar, és akkor, amikor akarja, még ha a követelmények a fejlesztés vége felé járva változnak is meg.

Az XP olyan bevált eljárásokon alapul, amelyeket együttműködésre szántak, bár sokan szemezgetnek az XP szabályai közül, és csak néhányat alkalmaznak belőlük. Az említett szabályok között ilyeneket találunk:

Készíts kis méretű változatokat, de gyakran!

A fejlesztést ismétlődő ciklusokban végezd!

Ne tegyél semmit a programba, ami nem szerepel a követelményleírásban (bármilyen csábító legyen is „a jövőnek szánt" szolgáltatásokat beépíteni)!

A tesztkódot írd meg *először*!

Felejtsd el a feszített ütemterveket – normál munkaidőben dolgozz!

Tervezd újra (javítsd fel) a kódot, ahol és amikor csak lehetőséget látsz rá!

Ne véglegesíts semmilyen kódot, amíg nem teljesít minden tesztet!

Készíts betartható ütemtervet, kis méretű kiadásokra alapozva!

Ragaszkodj az egyszerűséghez!

Programozz párban, és cserélgesd a párodat, hogy végül lényegében mindenki tudjon mindent a kódról!

A SimpleDotCom osztály tesztkódjának megírása

Olyan tesztkódot kell írnod, amely létre tud hozni egy SimpleDotCom objektumot, és képes futtatni annak tagfüggvényeit. A SimpleDotCom osztály esetében csak a *checkYourself()* tagfüggvény az érdekes a számodra, bár a *setLocationCells()* tagfüggvényt is meg kell majd valósítanod, hogy a *checkYourself()* tagfüggvény helyesen fusson.

Nézd meg jól a *checkYourself()* tagfüggvény alábbi kódvázlatát (a *setLocationCells()* pofonegyszerű beállítófüggvény, ezért nem kell aggódnod miatta, egy „valódi" alkalmazásban azonban összetettebb beállító tagfüggvényre lenne szükség, amit *célszerű* tesztelni), majd tedd fel magadnak a kérdést: „Ha megvalósítanám a checkYourself() tagfüggvényt, milyen tesztkódot kellene írnom, amely bizonyítja, hogy a tagfüggvény helyesen működik?"

Ennek a kódvázlatnak az alapján...

TAGFÜGGVÉNY: ***String checkYourself(String userGuess)***
KIOLVAS a felhasználó tippje String paraméterként
ÁTALAKÍT a felhasználó tippje ***int***-té
ISMÉTEL az ***int*** tömb minden helyzetcellájával
// ÖSSZEVET a felhasználó tippje a helyzetcellával
IF a felhasználó tippje egyezik
NÖVEL találatok száma
// KIDERÍT hogy az utolsó helyzetcella volt-e
IF találatok száma 3, **RETURN** „kill" mint eredmény
ELSE nem süllyedt, ezért **RETURN** „hit"
IF VÉGE
ELSE a felhasználó tippje nem egyezik, ezért **RETURN** „miss"
IF VÉGE
ISMÉTEL VÉGE
TAGFÜGGVÉNY VÉGE

...a tesztnek ezeket kell elvégeznie:

1. Példányosítania kell egy SimpleDotCom objektumot.
2. Hozzá kell rendelnie egy helyhez (egy 3 int-ből álló tömbhöz, például: {2,3,4}).
3. Létre kell hoznia egy String-et a felhasználó tippjének tárolására ("2", "0" stb.).
4. Meg kell hívnia a checkYourself() tagfüggvényt, átadva neki a hamis tippet.
5. Ki kell írnia az eredményt, hogy lásd, helyes-e a kód („passed" vagy „failed", vagyis „átment" vagy „megbukott").

kódvázlat | tesztkód | valódi kód

Nincsenek
hülye kérdések

K: Lehet, hogy lemaradtam valamiről, de pontosan hogyan is lehet tesztet futtatni valamin, ami még nem is létezik?

V: Nem maradt ki semmi. Nem mondtunk olyat, hogy a teszt *futtatásával* kezdjük – csak azt, hogy először *megírjuk* a tesztet. Amikor megírod a tesztkódot, még nincs semmid, amin lefuttathatnád, ezért valószínűleg le sem tudod fordítani, amíg el nem készítesz egy „kódcsonkot", ami a fordítást lehetővé teszi, de a tesztet kudarcra (például null visszaadására) ítéli.

K: Még most sem értem. Miért nem várjuk meg, amíg a kód elkészül, és írjuk meg az után a tesztkódot?

V: A tesztkód végiggondolása (és megírása) segít tisztázni, hogy magának a tagfüggvénynek mit kell csinálnia. Így mire a megvalósítás elkészül, már van egy tesztkódod, amely készen áll a megvalósítás ellenőrzésére. Ezenkívül *tudod*, hogy ha nem írod meg most, akkor *soha* nem fogod, mert mindig lesz valami, ami jobban izgat. Ideális esetben írsz egy kis tesztkódot, majd *csak* azt a megvalósítást készíted el, amire szükséged van ahhoz, hogy az adott teszt sikeresen lefusson. Ezt ismét egy *újabb* tesztkód megírása követi, majd az *ennek* a tesztnek a teljesítéséhez szükséges megvalósítás. Minden tesztkörben az *összes* korábban megírt tesztet lefuttatod, hogy megbizonyosodj róla, hogy az újonnan hozzáadott kódok nem teszik működésképtelenné a korábban már tesztelt kódrészeket.

```
public class SimpleDotComTestDrive {

   public static void main (String[] args) {

      SimpleDotCom dot = new SimpleDotCom();

      int[] locations = {2,3,4};

      dot.setLocationCells(locations);

      String userGuess = "2";

      String result = dot.checkYourself(userGuess);

      String testResult = "failed";

      if (result.equals("hit") ) {

           testResult = "passed";

       }

       System.out.println(testResult);

   }

}
```

példányosítunk egy SimpleDotCom objektumot

létrehozunk egy int tömböt a dotcom helyzetének (3 egymást követő int-nek a lehetséges 7-ből) tárolására

meghívjuk a beállító tagfüggvényt a dotcom-ra

megadunk egy hamis felhasználói tippet

meghívjuk a checkYourself() tagfüggvényt a dotcom objektumra, és átadjuk neki a hamis tippet

ha a hamis tipp (2) „találatot" ad vissza, akkor működik

kiírjuk a teszt eredményét („passed" vagy „failed")

Hegyezd ki a ceruzád!

A következő néhány oldalon megvalósítjuk a SimpleDotCom osztályt, majd visszatérünk a tesztosztályhoz. Nézd meg a fenti tesztkódot: mivel lehetne még kiegészíteni? Mit *nem* vizsgálunk ebben a kódban, amit *kellene*? Írd ide az ötleteidet (vagy a szerinted szükséges kódsorokat):

A checkYourself() tagfüggvény

A kódvázlat nem ültethető át egy az egyben Java-kódra; itt-ott igazítanod kell rajta. A kódvázlat sokkal jobb képet ad arról, hogy a kódnak *mit* kell csinálnia, de most meg kell találnod azt a Java-kódot, ami azt is leírja, *hogyan.*
Az agyad elrejtett zugaiban töprengj el azon is, hogy a kód mely részein érdemes (vagy szükséges) javítani. Az alább kis körben látható számok olyan dolgokat (nyelvtani elemeket és nyelvi szolgáltatásokat) jelölnek, amelyekkel még nem találkoztál. A magyarázatuk a következő oldalon olvasható.

KIOLVAS a felhasználó tippje
ÁTALAKÍT a felhasználó tippje ***int***-té

ISMÉTEL az ***int*** tömb minden helyzetcellájával
IF a felhasználó tippje egyezik

NÖVEL találatok száma

// KIDERÍT hogy az utolsó helyzetcella volt-e
IF találatok száma 3,
RETURN „kill" mint eredmény
ELSE nem süllyedt, ezért
RETURN „hit"
ELSE
RETURN „miss"

```
public String checkYourself(String stringGuess) {
    int guess = Integer.parseInt(stringGuess);   (1)  <- int-té alakítjuk a String-et
    String result = "miss";   <- létrehozunk egy változót az eredmény tárolására, amit visszaadunk. alapértelmezésként a „miss"-t adjuk meg (vagyis feltételezzük, hogy „nem talált")
    (2) for (int cell : locationCells) {   <- a locationCells tömb minden cellájára (az objektum minden cellahelyzetére) megismételjük
        if (guess == cell) {   <- összevetjük a felhasználó tippjét a tömbnek ezzel az elemével (cellájával)
            result = "hit";
            (3) numOfHits++;   <- találatunk van!
            (4) break;   <- kilépünk a ciklusból, a többi cellát nem kell vizsgálni
        } // if vége
    } // for vége
    if (numOfHits == locationCells.length) {
        result = "kill";   <- a ciklusból kint vagyunk, de nézzük, hogy „meghaltunk-e" (3-szori találat) – ha igen, az eredmény karakterláncát „kill"-re változtatjuk
    } // if vége
    System.out.println(result);   <- megjelenítjük az eredményt a felhasználó számára („miss", hacsak nem változtattuk „hit"-re vagy „kill"-re)
    return result;   <- az eredményt visszaadjuk a hívó tagfüggvénynek
} // a tagfüggvény vége
```

kódvázlat tesztkód valódi kód

Csak az újdonságok

Ezen az oldalon azokat a dolgokat láthatod, amelyekkel még nem találkoztál korábban. Nyugalom! A hiányzó részleteket megtalálod a fejezet végén; itt csak annyit mondunk el, amennyi a továbblépéshez szükséges.

1 **String int-té** alakítása

A Java egyik beépített osztálya.

Az Integer osztály egyik tagfüggvénye, amely tudja, hogyan kell az általa ábrázolt int-ként „értelmezni" egy String-et.

Egy String-et vár.

```
Integer.parseInt("3")
```

2 A **for** ciklus

Egy ciklus bevezetésekor ezt így olvashatod: „ismételd a 'locationCells' tömb minden elemére: fogd a tömb következő elemét, és rendeld hozzá a 'cell' int változóhoz".

A kettőspont (:) azt jelenti, hogy „-ban/ben", tehát ez az egész annyit tesz, hogy „a locationCells-BEN található minden int értékre...".

```
for (int cell : locationCells) { }
```

Egy változót vezet be, amely a tömb egy elemét fogja tárolni. Ebbe a változóba (ami az esetünkben egy „cell" nevű int változó) a ciklus minden ismétlésekor a tömb más-más eleme kerül, amíg az elemek el nem fogynak (vagy a kód végre nem hajt egy „break" utasítást...lásd a 4-es pontot alább).

A tömb, amit a ciklusnak végig kell járnia. A ciklus minden ismétlésekor a tömb következő elemét rendeled a „cell" változóhoz. (Erről bővebben a fejezet végén lesz szó.)

3 Az **utónövelő művelet**

A ++ azt jelenti, hogy az adott elemhez (akármi is legyen az) hozzáadunk 1-et (más szavakkal: 1-gyel növeljük)

```
numOfHits++
```

A numOfHits++ (ebben az esetben) ugyanazt jelenti, mint a numOfHits = numOfHits + 1, csak egy kicsit rövidebb (vagyis hatékonyabb).

4 A **break** utasítás

```
break;
```

Kiléptet a ciklusból, mégpedig itt és most. Nincs ismétlés, nincs logikai vizsgálat – kifelé, de rögtön!

Nincsenek

hülye kérdések

K: Mi történik az Integer.parseInt() tagfüggvényben, ha az átadott dolog nem szám? A függvény felismeri a betűzött számokat is (például: „three")?

V: Az Integer.parseInt() csak olyan karakterláncokkal működik, amelyek számjegyek (0,1,2,3,4,5,6,7,8,9) ASCII-értékét ábrázolják. Ha olyasmit próbálunk értelmeztetni vele, mint a „two" vagy a „szürcs", a kód futáskor kiakad. (*Kiakadás* alatt valójában azt értjük, hogy *kivételt vált ki*, de a kivételekről csak a *Kivételek* című fejezetben lesz szó, úgyhogy a *kiakad* egyelőre megfelel.)

K: A könyv elején egy olyan *for* ciklust mutattatok, ami nagyon különbözött ettől. A *for* ciklusoknak két különböző fajtája van?

V: Igen! A Java első változataiban még csak egyetlen típusa létezett a *for* ciklusoknak (a bővebb kifejtést lásd a fejezet későbbi részében), ami így nézett ki:

```
for(int i = 0;i < 10;i++){
// ezt ismételd 10-szer
}
```

Ezt az alakot bármilyen ciklushoz (loop) használhatod. Csakhogy...a Java 5.0-ben (Tiger) a *továbbfejlesztett* for ciklus (enhanced *for* loop – ez a hivatalos neve) is megjelent: ezt akkor alkalmazhatod, ha a ciklusnak egy tömb (vagy – ahogy a következő fejezetben látni fogod – egy *másik* fajta gyűjtemény) elemein kell végigmennie. A tömböket is bejárhatod a régi, bevált for ciklusokkal, de a *továbbfejlesztett* for ciklussal könnyebb.

A SimpleDotCom és SimpleDotComTester osztályok végleges kódja

```
public class SimpleDotComTestDrive {

    public static void main (String[] args) {
        SimpleDotCom dot = new SimpleDotCom();
        int[] locations = {2,3,4};
        dot.setLocationCells(locations);
        String userGuess = "2";
        String result = dot.checkYourself(userGuess);
    }
}
```

```
public class SimpleDotCom {

    int[] locationCells;
    int numOfHits = 0;

    public void setLocationCells(int[] locs) {
        locationCells = locs;
    }

    public String checkYourself(String stringGuess) {
        int guess = Integer.parseInt(stringGuess);
        String result = "miss";
        for (int cell : locationCells) {
            if (guess == cell) {
                result = "hit";
                numOfHits++;
                break;
            }
        } // ki a ciklusból

        if (numOfHits ==
            locationCells.length) {
            result = "kill";
        }
        System.out.println(result);
            return result;
    } // a tagfüggvény lezárása
} // az osztály lezárása
```

Van itt egy aprócska hiba. A kód lefordítható és futtatható, de néha... Egyelőre ne aggódj miatta, de később *muszáj* lesz szembenézni vele.

Mit kell látnod, amikor ezt a kódot futtatod?

A tesztkód létrehoz egy SimpleDotCom objektumot, és hozzárendeli a 2,3,4 helyzetet, majd átadja a hamis „2" felhasználói tippet a checkYourself() tagfüggvénynek. Ha a kód helyesen működik, ennek az eredménynek kell kiíródnia:

```
java SimpleDotComTestDrive
hit
```

kódvázlat tesztkód valódi kód

Hegyezd ki a ceruzád!

Felépítetted a tesztosztályt és a SimpleDotCom osztályt, magát a *játékot* azonban még nem készítetted el. Az előző oldalon található kód és a játék követelményleírása alapján írd le az ötleteidet a Game osztály kódvázlatához! Néhány sort beírtunk helyetted, hogy segítsünk az első lépések megtételében. A játék tényleges kódja a következő oldalon található, ezért ***ne lapozz addig, amíg el nem végezted ezt a feladatot!*** A vázlatnak körülbelül 12–18 sorból kell állnia (az általunk beírt sorokkal együtt, de a csak kapcsos zárójelet tartalmazó sorok *nélkül*).

TAGFÜGGVÉNY ***public static void main (String [] args)***

BEVEZET int változó a felhasználói tippek tárolására, *numOfGuesses* néven

KISZÁMÍT véletlenszám 0 és 4 között, amely a kezdőcella helyzetét adja meg

WHILE a dotcom még él:

KIOLVAS felhasználói tipp a parancsssorból

A SimpleDotComGame feladatai:

1. Létre kell hoznia az egyetlen SimpleDotCom objektumot.
2. El kell helyeznie az objektumot (a hét virtuális cellából álló egyetlen sor három egymást követő cellájában)
3. Kérnie kell egy tippet a felhasználótól.
4. Ellenőriznie kell a tippet.
5. Ezt addig kell ismételnie, amíg a dotcom el nem süllyed.
6. Ki kell írnia a felhasználónak, hogy hány lövésre volt szükség.

Egy teljes játék

```
File Edit Window Help Runaway
%java SimpleDotComGame
enter a number  2
hit
enter a number  3
hit
enter a number  4
miss
enter a number  1
kill
You took 4 guesses
```

A SimpleDotComGame osztály kódvázlata

Minden a main()-ben történik

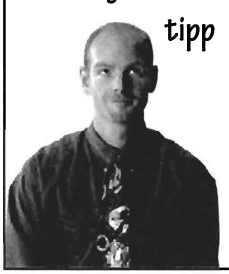
Vannak dolgok, amiket csak úgy, a bizalom jegyében kell elfogadnod. A kódvázlatunk egyik sora például azt mondja, hogy „KIOLVAS felhasználói tipp a parancsssorból". Hogy őszinték legyünk, ez egy kicsit több annál, mint amit a nulláról kiindulva most meg szeretnénk valósítani. Szerencsére azonban objektumközpontú tervezést folytatunk, ami azt jelenti, hogy megkérhetsz egy *másik* osztályt/objektumot, hogy tegyen meg neked valamit, anélkül, hogy foglalkoznod kellene azzal, hogy ***miként*** teszi. Amikor kódvázlatot írsz, feltételezned kell, hogy *valahogy* képes leszel elvégezni, amit meg kell tenned, így a teljes agyakapacitásodat a logika kidolgozására fordíthatod.

public static void main (String [] args)

BEVEZET int változó a felhasználói tippek tárolására, *numOfGuesses* néven, 0 alapértékkel
LÉTREHOZ új SimpleDotCom-példány
KISZÁMÍT véletlenszám 0 és 4 között, amely a kezdőcella helyzetét adja meg
LÉTREHOZ 3 int-ből álló int tömb a véletlenszám alapján, a számot 1-gyel, majd az így kapott eredményt 2-vel növelve (például: 3,4,5)
MEGHÍV *setLocationCells()* tagfüggvény a SimpleDotCom-példányra
BEVEZET logikai változó, amely a játék állapotát ábrázolja, *isAlive* („Él") néven. **BEÁLLÍT** true értékre

WHILE a dotcom még él (isAlive == true):
KIOLVAS felhasználói tipp a parancsssorból
// ELLENŐRIZ felhasználói tipp
MEGHÍV checkYourself() tagfüggvény a SimpleDotCom-példányra
NÖVEL numOfGuesses változó
// ELLENŐRIZ halott-e a dotcom
IF az eredmény „kill"
BEÁLLÍT *isAlive* false értékre (ami azt jelenti, hogy nem lépünk be újra a ciklusba)
KIÍR a felhasználó tippjeinek száma
IF VÉGE
WHILE VÉGE
TAGFÜGGVÉNY VÉGE

Metakognitív tipp



Egyszerre ne dolgoztasd túl hosszú ideig az agyad egyik felét. Ha 30 percnél tovább csak az agyad bal féltekéje dolgozik, az olyan, mintha csak a bal *karod* edzenéd fél óráig. Pihentesd mindkét agyféltekédet azzal, hogy rendszeres időközönként féltekét váltasz. Így amikor átkapcsolsz, a másik félteke pihenhet és regenerálódhat. A bal agyféltekét az olyan feladatok dolgoztatják meg, mint a lépésről lépésre haladó feladatok, a logikai feladványok megfejtése vagy az elemzés, míg a jobb féltekét a metaforák, a kreatív problémamegoldás, a mintaillesztés és a vizualizáció hozza működésbe.

Újratöltve

- A Java-programok fejlesztését egy magasszintű terv elkészítésével célszerű elkezdeni.
- Általában három dolgot kell megírnod, amikor új osztályt készítesz:
  kódvázlat
  tesztkód
  valódi (Java-) kód
- A kódvázlatnak azt kell leírnia, hogy *mit* kell csinálni, nem azt, hogy *hogyan.* A megvalósítás később következik.
- A kódvázlat segítségével tervezheted meg a tesztkódot.
- A tesztkódot a tagfüggvények megvalósítása *előtt* írd meg.
- Válaszd a *for* ciklusokat a *while* ciklusokkal szemben, ha tudod, hogy hányszor akarod megismételtetni a ciklus kódját.
- Ha egy változó értékét 1-gyel szeretnéd növelni (increment), használd az elő- vagy utó*növelő* műveletet (x++;).
- Ha egy változót értékét 1-gyel szeretnéd csökkenteni (decrement), használd az elő- vagy utó*csökkentő* műveletet (x--;).
- Egy String int értékét az `Integer.parseInt()` segítségével kaphatod meg.
- Az `Integer.parseInt()` csak akkor működik, ha a String egy számjegyet ábrázol („0", „1", „2" stb.)
- Egy ciklusból a *break* utasítással léphetsz ki a ciklus befejeződése előtt (vagyis akkor is, ha a logikai tesztfeltétel továbbra is igaz).



Üdvözlet Szellemvárosból!

A játék main() tagfüggvénye

Ahogy a SimpleDotCom osztály esetében is tetted, töprengj el azon, hogy a kód mely részein érdemes (vagy szükséges) javítani. Az alább kis körben látható számok olyan dolgokat jelölnek, amelyekre fel szerettük volna hívni a figyelmedet. A magyarázatuk a következő oldalon olvasható. Ha pedig azon morfondíroznál, hogy miért ugrottuk át a tesztkódot ennél az osztálynál… Nos, a játékhoz nem kell tesztosztály. Csak egy tagfüggvénye van – mégis, mit csinálnál a tesztkódban? Létrehoznál egy *külön* osztályt, ami meghívná a *main()*-t *erre* az osztályra? Mi nem fárasztottuk magunkat ezzel.

```
public static void main(String[] args) {
    int numOfGuesses = 0;
    GameHelper helper = new GameHelper();

    SimpleDotCom theDotCom = new SimpleDotCom();
    int randomNum = (int) (Math.random() * 5);

    int[] locations = {randomNum, randomNum+1, randomNum+2};

    theDotCom.setLocationCells(locations);
    boolean isAlive = true;

    while(isAlive == true) {
        String guess = helper.getUserInput("enter a number");
        String result = theDotCom.checkYourself(guess);
        numOfGuesses++;
        if (result.equals("kill")) {
            isAlive = false;
            System.out.println("You took " + numOfGuesses + " guesses");
        } // if vége
    } // while vége
} // main vége
```

BEVEZET változó a felhasználói tippek számának tárolására, 0 alapértékkel

létrehozunk egy változót a felhasználói tippek számának követésére

ez egy különleges osztály, amit mi írtunk. ez tartalmazza a felhasználói tippet beolvasó tagfüggvényt – tegyünk most úgy, mintha a Java része lenne

LÉTREHOZ SimpleDotCom-objektum

létrehozzuk a dotcom-objektumot

KISZÁMÍT véletlenszám 0 és 4 között

létrehozunk egy véletlenszámot az első cellához, és ezt alapul véve létrehozzuk a cellahelyek tömbjét

LÉTREHOZ int tömb a 3 cellahellyel

MEGHÍV setLocationCells() a dotcom-objektumra

meghatározzuk a dotcom cellahelyeit (a tömböt)

BEVEZET isAlive logikai változó

létrehozunk egy logikai változót, ami nyomon követi, hogy a dotcom még él-e, és ezt használjuk a while ciklus vizsgálatában. addig ismétlünk, amíg a dotcom él.

WHILE a dotcom még él

KIOLVAS felhasználói tipp

kiolvassuk a felhasználói tipp karakterláncát

// ELLENŐRIZ tipp

MEGHÍV checkYourself() a dotcom-ra

megkérjük a dotcom-ot a tipp ellenőrzésére, a kapott eredményt pedig egy String-be mentjük

NÖVEL numOfGuesses

növeljük a tippszámláló értékét

// ELLENŐRIZ halott-e a dotcom

IF az eredmény „kill"

„süllyedt"? – ha igen, az isAlive-ot false-ra állítjuk (így nem lépünk be újra a ciklusba), és kiírjuk a felhasználó tippjeinek számát

BEÁLLÍT *isAlive* false-ra

KIÍR a felhasználó tippjeinek száma

kódvázlat | tesztkód | valódi kód

random() és getUserInput()

Ezen az oldalon azt a két dolgot látod, ami egy kissé bővebb magyarázatra szorul. Csak annyit mondunk el, amennyi a továbblépéshez szükséges; a GameHelper osztályról további részleteket a fejezet végén találsz.

1 Véletlenszám létrehozása

Ez egy „típusátalakítás", ami arra kényszeríti a közvetlenül utána következő dolgot, hogy az itt (a zárójelben) megadott típust vegye fel. A Math.random egy double értéket ad vissza, ezért intté kell alakítanunk (egy szép egész számot szeretnénk 0 és 4 között). Esetünkben a típusátalakítás levágja a double érték tört részét.

A Math.random tagfüggvény egy nulla és „éppen kevesebb, mint 1" közötti számot ad vissza, így ez a képlet (a típusátalakítással) egy 0 és 4 (mármint 0 – 4,999..., egészre kerekítve) közötti számot eredményez.

```
int randomNum = (int) (Math.random() * 5)
```

Bevezetünk egy int változót a visszakapott véletlenszám tárolására.

A Java egyik beépített osztálya.

A Math osztály egyik tagfüggvénye.

2 A felhasználói tipp beolvasása a GameHelper osztály segítségével

Egy korábban létrehozott példány abból az osztályból, amelyet a játék segédjeként készítettünk. Az osztály neve GameHelper, és még nem találkoztál vele (de majd fogsz).

Ez a tagfüggvény egy String argumentumot vár, amelyet arra használ, hogy tippet kérjen a felhasználótól a parancssorban. Amit itt átadsz, az jelenik meg a parancsablakban, mielőtt a tagfüggvény keresni kezdené a bemenetet.

```
String guess = helper.getUserInput("enter a number");
```

Bevezetünk egy String változót a felhasználótól kapott String („3", „5" stb.) tárolására.

A GameHelper osztály tagfüggvénye, amely parancssori bemenetet kér a felhasználótól, beolvassa azt, miután a felhasználó lenyomta az ENTER billentyűt, és visszaadja az eredményt String-ként.

Még egy utolsó osztály: GameHelper

Elkészítetted a *dot com* osztályát.
Elkészítetted a *játék* osztályát.
Már csak a *segédosztály* maradt hátra – az, amelyikben a `getUserInput()` tagfüggvény található. Most azonban nem szeretnénk belemenni a parancssori bemenetet kérő kód boncolgatásába, mert túl sok olyan kérdést vetne fel, amivel célszerűbb később foglalkozni (a „később" úgy értendő, hogy a 14. fejezetben).

Csak másold le* az alábbi kódot, és add oda a fordítóprogramnak, hogy elkészítse belőle a GameHelper nevű osztályt. Dobd be mindhárom osztályt (SimpleDotCom, SimpleDotComGame, GameHelper) ugyanabba a könyvtárba, és tedd azt a munkakönyvtáraddá.

Amikor a  **Készre sütött kód** ikont látod, olyan kódot találsz alatta, amit úgy kell begépelned és elfogadnod, ahogy van. Bízz bennünk: *később* megtanulod majd, hogyan működik.





Készre sütött kód

```
import java.io.*;
public class GameHelper {
   public String getUserInput(String prompt) {
      String inputLine = null;
      System.out.print(prompt + "  ");
      try {
        BufferedReader is = new BufferedReader(
        new InputStreamReader(System.in));
       inputLine = is.readLine();
       if (inputLine.length() == 0 )  return null;
      } catch (IOException e) {
        System.out.println("IOException: " + e);
      }
      return inputLine;
   }
}
```

*Tudjuk, hogy nagyon szeretsz gépelni, de ha kivételesen mégis inkább mást csinálnál, a készre sütött kódot elérhetővé tettük a wickedlysmart.com címen is.

Játsszunk!

Ez történik, ha elindítod a játékot, és beírod az 1,2,3,4,5,6 számokat. Jól néz ki.

Egy teljes játék

(nálad több vagy kevesebb üzenet is lehet)

```
File Edit Window Help Smile
%java SimpleDotComGame
enter a number  1
miss
enter a number  2
miss
enter a number  3
miss
enter a number  4
hit
enter a number  5
hit
enter a number  6
kill
You took 6 guesses
```

Mi ez? Egy hiba?

Uramisten!

Ez történik, ha azt írod be, hogy 1,1,1:

Egy másfajta játék

(jaj)

```
File Edit Window Help Faint
%java SimpleDotComGame
enter a number  1
hit
enter a number  1
hit
enter a number  1
kill
You took 3 guesses
```



Hegyezd ki a ceruzád!

Micsoda izgalmak!

Megtaláljuk a hibát?

Kijavítjuk a hibát?

Maradj velünk a következő fejezetre is, ahol választ kapsz ezekre a kérdésekre – és még sokkal többre is…

Addig is, próbáld kitalálni, hogy mi siklott félre, és hogyan lehetne kijavítani a hibát!

További tudnivalók a for ciklusokról

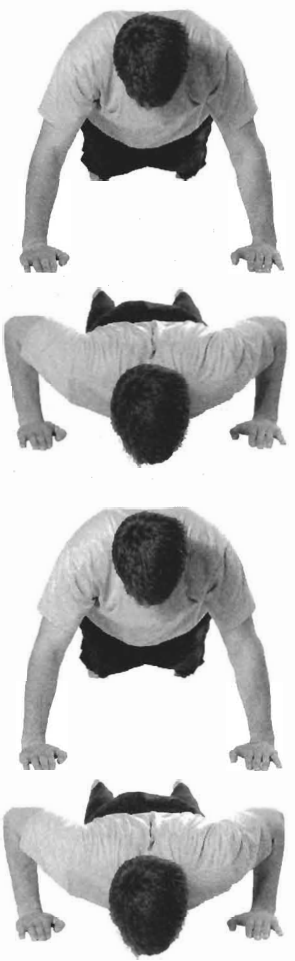
Az *ebben* a fejezetben elkészíteni kívánt játék kódját teljes egészében áttekintettük (de a következő fejezetben visszatérünk rá, hogy befejezzük a játék deluxe változatát). Nem akartuk megszakítani a munkád mindenféle apró-cseprő részlettel és háttérinformációval, ezért ide tettük őket. Először a for ciklusokat vesézzük ki bővebben – ha C++-programozó vagy, ezt az utolsó néhány oldalt elég felületesen átlapoznod...

Normál (nem továbbfejlesztett) for ciklusok

```
for (int i = 0; i < 100; i++) { }
```

- `int i = 0` – előkészítés
- `i < 100` – logikai vizsgálat
- `i++` – utónövelő művelet (cikluskifejezés)
- `{ }` – ide jön az ismétlődő kód (a törzs)

Ismételd 100-szor



A fenti kód magyarul azt jelenti, hogy „Ismételd 100-szor".
A fordító pedig így értelmezi:

- hozz létre egy *i* változót, és állítsd az értékét 0-ra
- ismételj, amíg *i* kisebb, mint 100
- a ciklus minden ismétlésének végén adj 1-et az *i*-hez

Első rész: *előkészítés*

Ez a rész arra szolgál, hogy bevezess és kezdőértékkel láss el egy változót, amelyet a ciklus törzsében fogsz használni, általában számlálóként. Itt valójában egynél több változót is előkészíthetsz, de erről majd a könyv későbbi részében beszélünk.

Második rész: *logikai vizsgálat*

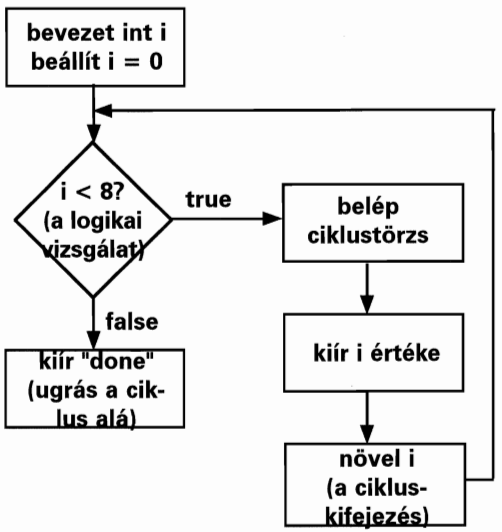
Ide kerül a feltételvizsgálat. Akármi is álljon itt, egy logikai értékké (tudod, ***true*** vagy ***false***) *kell* feloldani azt. Alkalmazhatsz vizsgálatot (például: x >= 4), de egy tagfüggvényt is meghívhatsz itt, amely egy logikai értéket ad vissza.

Harmadik rész: *cikluskifejezés*

Ebbe a kódrészbe egy vagy több olyan műveletet tehetsz, amelyet végre szeretnél hajtatni a ciklus minden ismétlésekor (iteration). Ne feledd, hogy ezeknek a műveleteknek a végrehajtására mindig a ciklus *végén* kerül sor.

Variációk ciklusokra

```
for (int i = 0; i < 8; i++) {
   System.out.println(i);
}
System.out.println("done");
```



A for és a while közötti különbség

A *while* ciklusokban csak logikai vizsgálat található; beépített kezdőérték-adó vagy cikluskifejezés nincs bennük. *While* ciklust akkor érdemes használnod, ha nem tudod, hányszor kell ismételni a ciklust, csak folytatni szeretnéd, amíg valamilyen feltétel teljesül. Ha viszont *tudod*, hányszor akarsz ismételni (például egy tömb hosszával megegyező alkalommal, 7-szer stb.), egy *for* ciklus világosabb lehet. Íme a fenti ciklus *while* használatára átírva:

```
int i = 0;
while (i < 8) {
   System.out.println(i);
   i++;
}
System.out.println("done");
```

be kell vezetned és elő kell készítened a számlálót

növelned kell a számláló értékét

kimenet:

```
File Edit Window Help Repeat
%java Test
0
1
2
3
4
5
6
7
done
```

++ --

Elő- és utónövelő/csökkentő műveletek

Így rövidítheted azt, hogy 1-et hozzáadsz egy változóhoz:

```
x++;
```

Ez megegyezik ezzel:

```
x = x + 1;
```

Az „adj 1-et az x jelenlegi értékéhez" vagy ***„növeld*** x-et 1-gyel" környezetében a fenti két utasítás ugyanazt jelenti.
Ez pedig...

```
x--;
```

...ugyanaz, mint ez:

```
x = x - 1;
```

A teljes kép soha nem csak ennyi. A műveleti jel (operátor) elhelyezése (akár a változó után, akár az előtt) hatással lehet az eredményre. Ha a műveleti jelet a változó *elé* teszed (előnövelés; például: ++x), az azt jelenti, hogy *„először* növeld x-et 1-gyel, *aztán* használd fel az x új értékét". Ez csak akkor számít, ha a ++x nagyobb kifejezés (expression) része, és nem önálló utasítás.

```
int x = 0;      int z = ++x;
```

Ennek lesz az eredménye: x = 1, z = 1.
Ha viszont a ++ jelet az x *után* írod (utónövelés), más eredményt kapsz:

```
int x = 0;      int z = x++;
```

Ennek lesz az eredménye: x = 1, de ***z = 0***, mert z előbb kapja meg az x értékét, és az x növelése csak ez *után* történik.

A továbbfejlesztett for ciklus

A Java 5.0 (Tiger) óta a Java nyelvben egy második fajta *for* ciklus is megtalálható: az úgynevezett *továbbfejlesztett for* (*enhanced for*), amely a tömbök és más gyűjtemények elemeinek (element) bejárását könnyíti meg. (A *másfajta* gyűjteményekről a következő fejezetben fogsz tanulni.) A továbbfejlesztett for tényleg csak ennyit nyújt: egy egyszerűbb módot arra, hogy végigmenj egy gyűjtemény minden elemén – de mivel a *for* ciklusnak ez a legelterjedtebb felhasználási módja, érdemes volt kibővíteni vele a nyelvet. A *továbbfejlesztett for ciklusra* még visszatérünk a következő fejezetben, amikor majd azokat a gyűjteményeket tárgyaljuk, amelyek *nem* tömbök.

Bevezetünk egy ciklusváltozót, amely a tömb egyetlen elemét fogja tárolni.

A kettőspont (:) azt jelenti, hogy „-ban/ben".

ide jön az ismétlődő kód (a törzs)

```
for (String name : nameArray) { }
```

A tömb elemeinek összeegyeztethetőnek kell lenniük a változó bejelentett típusával.

A „name" változóba a ciklus minden ismétlésekor a tömb más-más eleme kerül

Az elemgyűjtemény, amit a ciklusnak végig kell járnia. Tételezzük fel, hogy a kód korábban valahol ezt mondta:
String[] nameArray = {"Fred", "Mary", "Bob"};
Az első ismétlésnél a name változó értéke "Fred", a második körben "Mary", és így tovább.

A fenti kód hétköznapi nyelven annyit tesz, hogy „a nameArray minden elemének esetében rendeld az elemet a 'name' változóhoz, és futtasd le a ciklus törzsét".

A fordító pedig így értelmezi:

- Hozz létre egy *name* nevű String változót, és állítsd az értékét null-ra.
- Rendeld a *nameArray* első értékét a name változóhoz.
- Futtasd le a ciklus törzsét (a kapcsos zárójelek által határolt kódblokkot).
- Rendeld a *nameArray* következő értékét a name változóhoz.
- Ezt addig ismételd, amíg *még vannak elemek a tömbben.*

Megjegyzés: attól függően, hogy korábban milyen programozási nyelvvel dolgoztak, egyesek úgy hívják a továbbfejlesztett for-t, hogy „for each" vagy „for in" ciklus, mert ez a jelentése: „minden egyes (EACH) dologra a gyűjteményben (IN)...".

Első rész: a *ciklusváltozó bevezetése*

Ez a rész arra szolgál, hogy bevezess és kezdőértékkel láss el egy változót, amelyet a ciklus törzsében fogsz használni. Ebbe a változóba a ciklus minden ismétlésekor más-más elem kerül a gyűjteményből. A változó típusának összeegyeztethetőnek kell lennie a tömb elemeivel, például egy *String[]* típusú tömbhöz nem vezethetsz be *int* típusú ciklusváltozót.

Második rész: a *tényleges gyűjtemény*

Ennek egy tömbre vagy más gyűjteményre mutató hivatkozásnak kell lennie. Ezúttal sem kell még foglalkoznod a gyűjtemények *más*, nem tömb típusú fajtáival – a következő fejezetben azokra is sor kerül.

String átalakítása int-té

```
int guess = Integer.parseInt(stringGuess);
```

A felhasználó beírja a tippjét a parancssorba, amikor a játék kéri. A tipp egy karakterlánc formájában érkezik ("2", "0" stb.), a játék pedig ezt a String-et adja át a checkYourself() tagfüggvénynek. A cellahelyzetek azonban egyszerű int-ek egy tömbben, márpedig egy int-et nem hasonlíthatsz össze egy String-gel.

Ez például ***nem működik***:

```
String num = "2";
int x = 2;
if(x == num) // szörnyű robbanás!
```

Ha ezt megpróbálod lefordíttatni, a fordítóprogram kiröhög:

```
operator == cannot be applied to
      int,java.lang.String
    if (x == num) { }
         ^
```

A szezon és a fazon problémájának elkerüléséhez a "2" *String*-ből a 2 *int* értéket kell előállítanod. A Java osztálykönyvtárában találsz egy Integer nevű beépített osztályt (igen, *osztályt*, és nem az int *alaptípust*), amelynek az egyik feladata a számokat *ábrázoló* karakterláncok átalakítása *valódi* számokká.

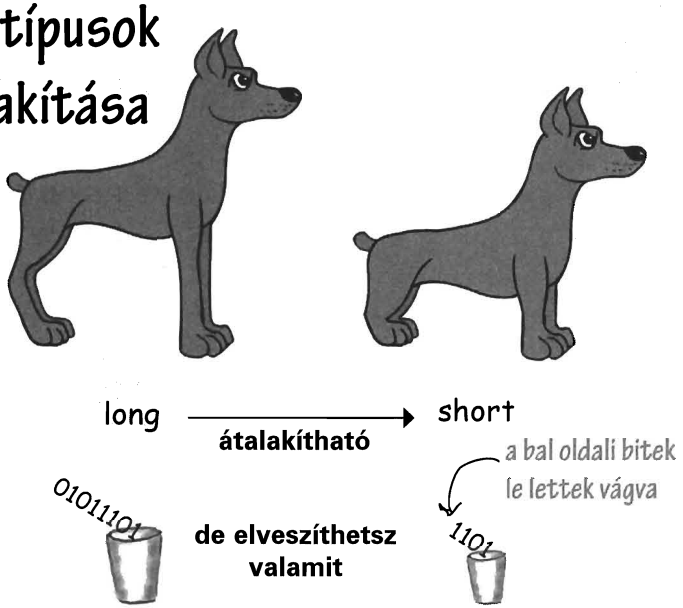
a Java egyik beépített osztálya

egy String-et vár

```
Integer.parseInt("3")
```

az Integer osztály egyik tagfüggvénye, amely tudja, hogyan kell az általa ábrázolt int-ként „értelmezni" egy String-et

Alaptípusok átalakítása



A 3. fejezetben beszéltünk a különféle alaptípusok (primitívek) méretéről, és hogy egy nagy dolgot nem gyömöszölhetsz bele egy kicsibe:

```
long y = 42;
int x = y; // nem fordítható le
```

A *long* típus nagyobb, mint az *int*, a fordító pedig nem tudhatja, hogy az adott *long* hol volt. Lehet, hogy kocsmázott a többi long-gal, és igen nagy értékeket döntött magába. Ha arra akarod kényszeríteni a fordítót, hogy egy nagyobb primitív változó értékét egy kisebbe tömje bele, a **típusátalakítás** vagy **típuskényszerítés** (cast) műveleti jelét kell alkalmaznod, ami így fest:

```
long y = 42;      // eddig jó
int x = (int) y; // x = 42, király!
```

A típuskényszerítés jelzése arra utasítja a fordítót, hogy fogja az y értékét, faragja le int méretűre, és rendelje az x-hez, ami maradt. Ha az y értéke nagyobb volt az x lehetséges legnagyobb értékénél, akkor egy furcsa (de kiszámítható*) szám marad:

```
long y = 40002;
// a 40002 meghaladja a short típus
// 16 bites korlátját
short x = (short) y;    // az x értéke most -25534!
```

A lényeg az, hogy a fordító megengedi, hogy ezt tedd. Tegyük fel, hogy egy lebegőpontos szám egész (*int*) részét szeretnéd megkapni:

```
float f = 3.14f;
int x = (int) f; // az x értéke 3 lesz
```

Arról viszont ne is *álmodj*, hogy bármit boolean értékké alakítasz, vagy fordítva!

*Előjelbitek, binárisok, „kettes komplemens" műveletek és más gyíkságok kellenek hozzá, amelyekről a B függelék elején lesz szó.



Gyakorlat

Légy a Java virtuális gép!



Az ezen az oldalon található Java-fájlok mindegyike egy-egy teljes forrásfájl. A te feladatod az, hogy eljátszd a Java virtuális gép (JVM) szerepét, és kiderítsd, hogy milyen kimenetet kapsz, ha futtatod a programot.

```
class Output {

  public static void main(String [] args) {
    Output o = new Output();
    o.go();
  }

  void go() {
    int y = 7;
    for(int x = 1; x < 8; x++) {
      y++;
      if (x > 4) {
         System.out.print(++y + " ");
      }
      if (y > 14) {
        System.out.println(" x = " + x);
        break;
      }
    }
  }
}
```

```
File Edit Window Help OM
% java Output
12 14
```

– vagy –

```
File Edit Window Help Incense
% java Output
12 14 x = 6
```

– vagy –

```
File Edit Window Help Believe
% java Output
13 15 x = 6
```



Gyakorlat

Kódmágnesek



Egy Java-program kódja összekeveredett a hűtőmágneseken. Össze tudod rakni úgy a kódrészleteket, hogy előálljon belőlük a lap alján látható kimenetet eredményező Java-program? Néhány kapcsos zárójel leesett a padlóra, és túl kicsi volt ahhoz, hogy fel tudjuk venni, úgyhogy nyugodtan egészítsd ki a kódot annyi kapcsos zárójellel, amennyit csak szükségesnek érzel!

```
x++;
```

```
if (x == 1) {
```

```
System.out.println(x + " " + y);
```

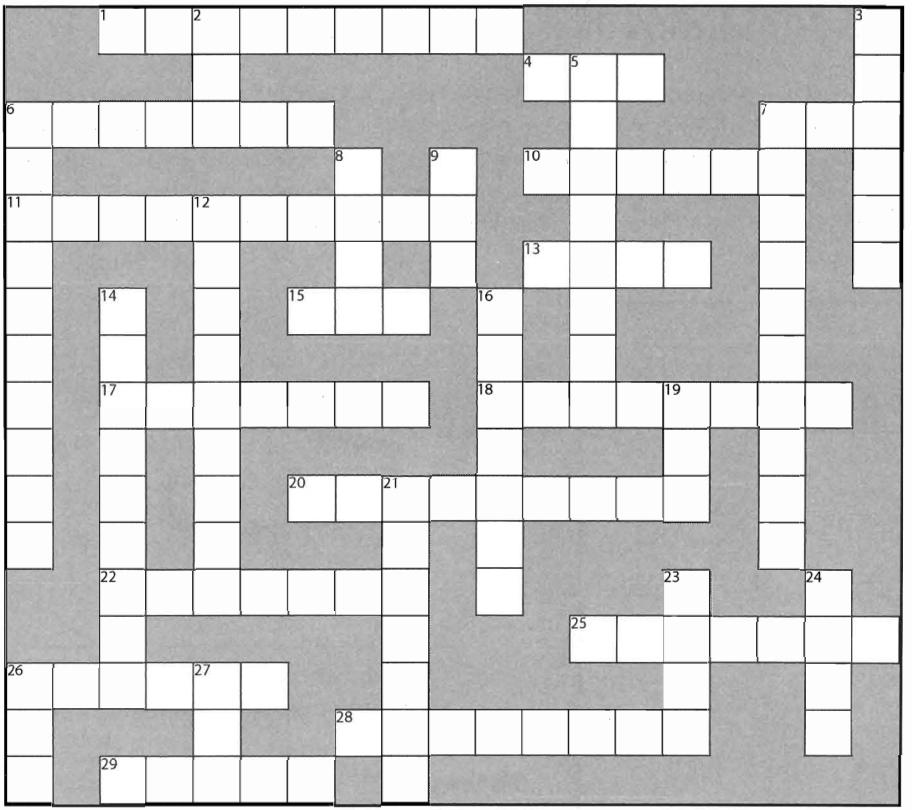
```
class MultiFor {
```

```
for(int  y = 4; y > 2; y--) {
```

```
for(int x = 0; x < 4; x++) {
```

```
public static void main(String [] args) {
```

```
File  Edit  Window  Help  Raid
% java MultiFor
0 4
0 3
1 4
1 3
3 4
3 3
```





Java-keresztrejtvény

Hogyan segíthet egy keresztrejtvény a Java megtanulásában? Nos, a rejtvényben szereplő szavak **mindegyike** a Javával kapcsolatos, ezenkívül a meghatározások metaforákat, szójátékokat és hasonlókat tartalmaznak. Ezek az elmecsavarok szerteágazó utakat égetnek a Java felé közvetlenül az agyadba!

Vízszintes

1. Megvalósít, számítógépes zsargonban (angolul)
4. Többrészes ciklus
6. Először tesztelő (programozási módszer, angolul)
7. 32 bit
10. Egy tagfüggvény válasza
11. Kódvázlatszerűség (angolul)
13. Típuskényszerítés angolul
15. A nagy eszköztár
17. Tömbegység (angolul)
18. Példány- vagy helyi lehet (angolul)
20. Automatizált eszközkészlet
22. Olyan, mint az egyik alaptípus, de nem az
25. Nem lehet rajta típuskényszerítést végezni
26. A Math osztály egyik tagfüggvénye
28. Átalakító tagfüggvény
29. Idő előtt kilép

Függőleges

2. Elő- (angolul)
3. Az osztályok igáslova (angolul)
5. Művelet(i jel), angolul
6. Kifejezés, angolul
7. Előkészít, kezdőértéket ad (angolul)
8. while vagy for
9. Beállít (mondjuk egy példányváltozót), angolul
12. Csökkent, angolul
14. Ciklus, másképp (és angolul)
16. Beszédes csomag
19. A tagfüggvények postása (rövidítve)
21. Nem igazi (angolul)
23. Utó- (angolul)
24. Itt lakik a Pi
26. Fordítsd le és ___ (persze angolul)
27. Ennyit jelöl a ++



Összekevert ÜZENETEK

Nézd meg az alábbi rövid Java-programot. A program egyik kódblokkja hiányzik. A feladatod az, hogy **összepárosítsd az alábbi kódjelölteket** (balra) azzal a **kimenettel**, amit akkor látnál, ha az adott blokkot szúrnád be a kódba. Nem minden kimenetsornak van párja, viszont egyes kimenetsorok többször is felhasználhatók. Kösd össze vonalakkal a kódjelölteket és a nekik megfelelő parancssori kimeneteket!

```
class MixFor5 {
  public static void main(String [] args) {
    int x = 0;
    int y = 30;
    for (int outer = 0; outer < 3; outer++) {
      for(int inner = 4; inner > 1; inner--) {

        [          ]   <-- ide jön a kódjelölt

        y = y - 2;
        if (x == 6) {
          break;
        }
        x = x + 3;
      }
      y = y - 2;
    }
    System.out.println(x + " " + y);
  }
}
```

Kódjelöltek:

```
x = x + 3;
```

```
x = x + 6;
```

```
x = x + 2;
```

```
x++;
```

```
x--;
```

```
x = x + 0;
```

Lehetséges kimenet:

```
45 6
```

```
36 6
```

```
54 6
```

```
60 10
```

```
18 6
```

```
6 14
```

```
12 14
```

párosíts össze minden kódjelöltet a neki megfelelő parancssori kimenettel



Gyakorlat – Megoldás

Légy a Java virtuális gép!

```
class Output {

  public static void main(String [] args) {
    Output o = new Output();
    o.go();
  }
  void go() {
    int y = 7;
    for(int x = 1; x < 8; x++) {
      y++;
      if (x > 4) {
         System.out.print(++y + " ");
      }
      if (y > 14) {
        System.out.println(" x = " + x);
        break;
      }
    }
  }
}
```

Figyelembe vetted a break utasítást? Milyen hatással van a kimenetre?

```
File Edit Window Help MotorcycleMaintenance
% java Output
13 15 x = 6
```

Kódmágnesek:

```
class MultiFor {

  public static void main(String [] args) {

    for(int x = 0; x < 4; x++) {

      for(int y = 4; y > 2; y--) {
        System.out.println(x + " " + y);
      }

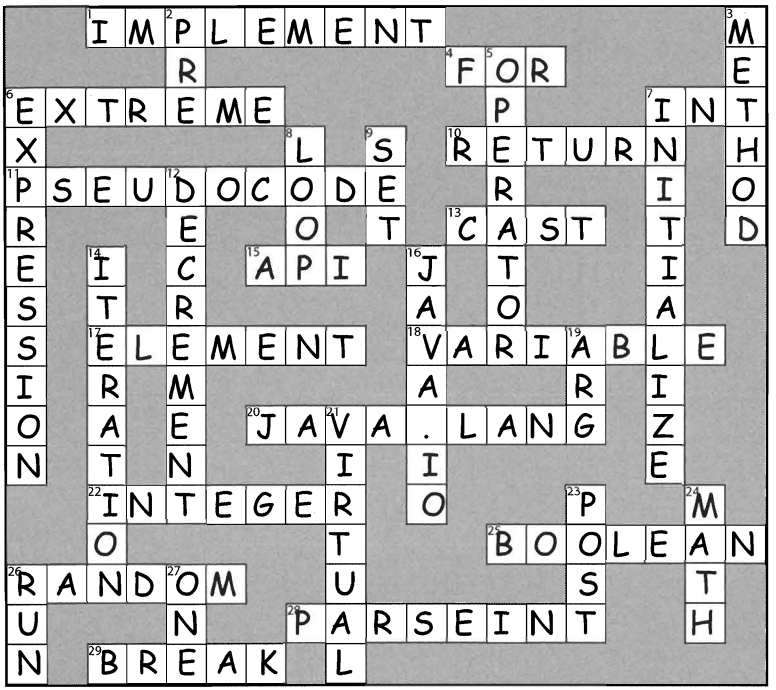
      if (x == 1) {
        x++;
      }
    }
  }
}
```

Mi történne, ha ez a kódblokk az „y" for ciklus elé kerülne?

```
File Edit Window Help Monopole
% java MultiFor
0 4
0 3
1 4
1 3
3 4
3 3
```



Fejtörők – Megoldás



Kódjelöltek:

```
x = x + 3;
x = x + 6;
x = x + 2;
x++;
x--;
x = x + 0;
```

Lehetséges kimenet:

```
45 6
36 6
54 6
60 10
18 6
6 14
12 14
```

A Java-könyvtár használata



A Javába osztályok százait építették be előre. Nem kell újra feltalálnod a kereket, ha tudod, hogyan találhatod meg, amire szükséged van, a Java-könyvtárban, más néven a **Java API**-ban (Application Programming Interface, alkalmazás-programozási felület). *Van jobb dolgod is.* Ha kódot akarsz írni, megteheted, hogy *csak* azokat a részeket írod meg, amelyek tényleg az alkalmazásod sajátjai. Ismersz olyan programozókat, akik mindennap pontban 5 órakor távoznak? És be se *jönnek* délelőtt 10-ig? Nos, **ők a Java API-t használják**, és nyolc oldallal később te is közéjük fogsz tartozni. A központi Java-könyvtár egy gigantikus osztályhalom, ami csak arra vár, hogy építőkockákként használd, hogy nagyrészt kész kódokból összeszereld a programodat. Az ebben a könyvben szereplő „készre sütött Java-kódok" olyan kódrészek, amelyeket nem neked kell a semmiből előteremtened, csak be kell gépelned. Sőt, a Java API tele van olyan kódokkal is, amelyek még *begépelést* sem igényelnek. Csak annyi a dolgod, hogy megtanuld használni őket.

Az előző fejezet függőben maradt izgalmakkal ért véget – nem javítottunk ki egy hibát

Ilyennek kellene lennie

Ez történik, ha elindítod a játékot, és beírod az 1,2,3,4,5,6 számokat. Jól néz ki.

Egy teljes játék

(nálad több vagy kevesebb üzenet is lehet)

```
File Edit Window Help Smile
%java SimpleDotComGame
enter a number  1
miss
enter a number  2
miss
enter a number  3
miss
enter a number  4
hit
enter a number  5
hit
enter a number  6
kill
You took 6 guesses
```

És így fest a hiba

Ez történik, ha azt írod be, hogy 2,2,2:

Egy másfajta játék

(jaj)

```
File Edit Window Help Faint
%java SimpleDotComGame
enter a number  2
hit
enter a number  2
hit
enter a number  2
kill
You took 3 guesses
```

A játék jelenlegi változatában elég egy találat, mert ha egyszerűen megismétled még kétszer, a hajó elsüllyed!

Mi is történik itt valójában?

```
public String checkYourself(String stringGuess) {

    int guess = Integer.parseInt(stringGuess);

    String result = "miss";

    for (int cell : locationCells) {

      if (guess == cell) {

        result = "hit";

        numOfHits++;

        break;

      } // if vége

    } // for vége

    if (numOfHits == locationCells.length) {

      result = "kill";

    } // if vége

    System.out.println(result);

    return result;

} // tagfüggvény vége
```

int-té alakítjuk a String-et.

Létrehozunk egy változót az eredmény tárolására, amit visszaadunk. Alapértelmezésként a „miss"-t adjuk meg (vagyis feltételezzük, hogy „nem talált").

A tömb minden elemére megismételjük.

Összevetjük a felhasználó tippjét a tömbnek ezzel az elemével (cellájával).

találatunk van!

Kilépünk a ciklusból, a többi cellát nem kell vizsgálni.

A ciklusból kint vagyunk, de nézzük, hogy „meghaltunk-e" (3-szori találat) – ha igen, az eredmény karakterláncát „kill"-re változtatjuk.

Megjelenítjük az eredményt a felhasználó számára („miss", hacsak nem változtattuk „hit"-re vagy „kill"-re).

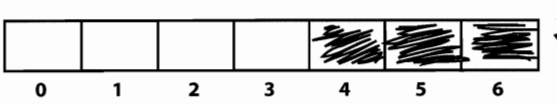
Az eredményt visszaadjuk a hívó tagfüggvénynek.

Itt van a bibi. Minden alkalommal találatot rögzítünk, amikor a játékos kitalálja egy cella helyét, még akkor is, ha az a cella már korábban is találatot kapott!

Valahonnan tudnunk kellene, hogy amikor a játékos lövése talál, olyan cellát ér-e, ami már korábban is találatot kapott. Ha igen, nem szabad találatként számolnunk.

Hogyan javítsuk ki a hibát?

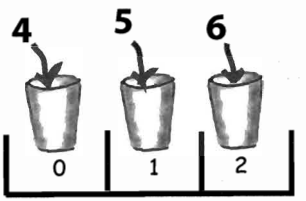
Kell egy módszer annak a kiderítésére, hogy egy cella korábban már kapott-e találatot. Végignézzük a lehetőségeket, de először vegyük sorra, hogy eddig mit tudunk...
Van egy 7 cellából álló virtuális sorunk, amelyben egy DotCom három egymást követő cellát foglal el. Az alábbi virtuális sor egy olyan DotCom-ot mutat, amely a 4,5,6 cellahelyeken terpeszkedik.



A virtuális sor a DotCom objektum 3 cellahelyével.

A DotCom-nak van egy példányváltozója – egy int tömb –, amely a DotCom objektum cellahelyeit tárolja.

locationCells (a DotCom példányváltozója)



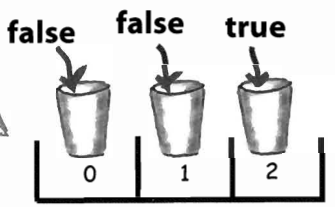
A DotCom cellahelyeit tároló tömb típusú példányváltozó. Ez a DotCom a 4,5,6 értékeket tárolja – ezeket a számokat kell a játékosnak kitalálnia.

1 Első lehetőség

Létrehozhatsz egy második tömböt, és minden alkalommal, amikor a felhasználó találatot ér el, a találatot elraktározhatod ebben a második tömbben. Amikor aztán a DotCom ismét találatot kap, ellenőrzöd ezt a tömböt, hogy kiderítsd, hogy az adott cellát nem érte-e már lövés.

Ha a tömb egy adott sorszámú helyén „true" áll, az azt jelenti, hogy a MÁSIK tömbnek (locationCells) ugyanezt a celláját már érte találat.

hitCells tömb (ez a DotCom új, boolean típusú tömb-példányváltozója lenne)



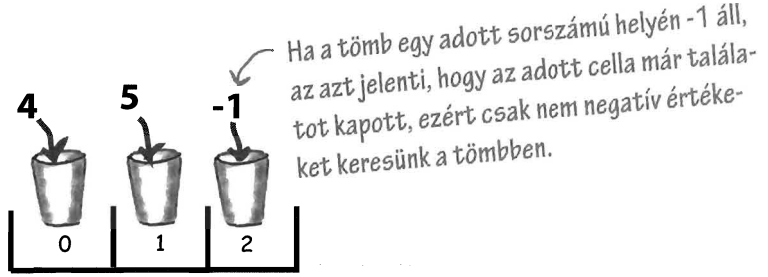
Ez a tömb három értéket tárol, amelyek a DotCom locationCells tömbjében található cellák „állapotát" jelképezik. Például ha a 2-es sorszámú cella találatot kap, akkor a „hitCells" tömb 2. elemét „true"-ra állítod.

Az első megoldás túl ormótlan

Az első megoldás a vártnál több munkát jelentene: minden alkalommal, amikor a felhasználó találatot ér el, meg kell változtatnod a *második* tömb (a „hitCells" tömb) állapotát – de először még ELLENŐRIZNED is kell a „hitCells" tömböt, hogy az adott cella nem kapott-e már korábban is találatot. A dolog persze működik, de lennie kell jobb megoldásnak…

② Második lehetőség

Megtarthatod az egyetlen, eredeti tömböt, ha minden kilőtt cella értékét –1-re állítod. Így csak EGY tömböt kell ellenőrizned és kezelned.



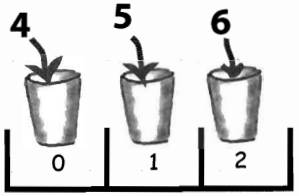
A második megoldás már jobb egy kicsit, de még mindig elég ormótlan

A második megoldás kevésbé esetlen, mint az első, de még mindig nem túl hatékony. Ugyanúgy végig kell járnod a tömb mindhárom rekeszét, akkor is, ha egy vagy több cella már érvénytelen (mert találat érte, és –1 az értéke). Lennie kell jobb megoldásnak…

③ Harmadik lehetőség

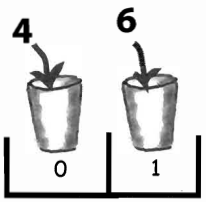
Törölhetsz minden cellahelyet, amint találatot kap, aztán módosíthatod a tömböt, hogy kisebb legyen. A tömbök viszont nem tudják változtatni a méretüket, ezért egy **új** tömböt kell létrehoznod, és a fennmaradó cellákat a régi tömbből át kell másolnod az újba.

locationCells tömb
(MIELŐTT egyetlen cellát is találat érne)



A tömb mérete kezdetben 3. Végigmegyünk mind a három cellán (a tömbpozíciókon), hogy egyezést keressünk a felhasználó tippje és a cella értéke (4,5,6) között.

locationCells tömb
(MIUTÁN a tömb 1-es sorszámú helyén álló „5" értékű cellát találat érte)



Amikor az „5" értékű cella találatot kap, létrehozunk egy új, kisebb tömböt, amelybe csak a fennmaradó cellahelyek kerülnek, és hozzárendeljük az eredeti locationCells hivatkozáshoz.

A harmadik megoldás sokkal jobb lenne, ha a tömb össze tudna zsugorodni, hogy ne kelljen egy új, kisebb tömböt létrehozni, belemásolni a fennmaradó értékeket, és újra elvégezni a hivatkozás hozzárendelését.

A checkYourself() tagfüggvény egy részének eredeti kódvázlata:

ISMÉTEL az ***int*** tömb minden helyzetcellájával
  // *ÖSSZEVET* a felhasználó tippje a helyzetcellával
  IF a felhasználó tippje egyezik
    NÖVEL találatok száma
    // *KIDERÍT* hogy az utolsó helyzetcella volt-e
    IF találatok száma 3, **RETURN** „kill"
    ELSE nem süllyedt, ezért **RETURN** „hit"
    IF VÉGE
  ELSE a felhasználó tippje nem egyezik, ezért **RETURN** „miss"
  IF VÉGE
ISMÉTEL VÉGE

Az élet csodaszép lenne, ha ezt erre módosíthatnánk:

ISMÉTEL az ***int*** tömb minden ***fennmaradó*** helyzetcellájával
  // *ÖSSZEVET* a felhasználó tippje a helyzetcellával
  IF a felhasználó tippje egyezik
    TÖRÖL az adott cellát a tömbből
    // *KIDERÍT* hogy az utolsó helyzetcella volt-e
    IF a tömb most üres, **RETURN** „kill"
    ELSE nem süllyedt, ezért **RETURN** „hit"
    IF VÉGE
  ELSE a felhasználó tippje nem egyezik, ezért **RETURN** „miss"
  IF VÉGE
ISMÉTEL VÉGE


Bárcsak találnék egy tömböt, ami képes összezsugorodni, amikor eltávolítok belőle valamit. Olyat, amin nem kell végigmenni, és megvizsgálni minden elemet, hanem megkérdezhetem tőle, hogy benne van-e az, amit keresek. Olyat, ami megengedné, hogy kivegyek belőle valamit, anélkül, hogy pontosan tudnom kellene, melyik rekeszben van. Meseszép lenne.
De tudom, hogy ez csak álom...

Ébredj, és érezd a könyvtár illatát!

Csoda történt! *Létezik* ilyen dolog.

Csak éppen nem tömbnek hívják, hanem *tömblistának (ArrayList)*.

A központi Java-könyvtár (az API) egyik osztálya.

A Java normál kiadása (Standard Edition; ez van neked is, hacsak nem az apró eszközökhöz szánt Micro Editionnel dolgozol – de hidd el, *arról tudnál*) előre felépített osztályok százait tartalmazza. Ezek olyanok, mint a mi készre sütött kódjaink, de a beépített osztályok már le is vannak fordítva.

Ez azt jelenti, hogy nem kell gépelni.

Csak használni őket.

Egy a Java-könyvtár számtalan osztálya közül.

A kódjaidban ugyanúgy használhatod, mintha te írtad volna.

(Megjegyzés: az add(Object elem) tagfüggvény igazából kicsit furcsább annál, mint amit itt látsz... az igazi tagfüggvényhez egy kicsit később érünk el a könyvben. Egyelőre elég, ha úgy gondolsz rá, mint egy add() tagfüggvényre, amelynek a hozzáadni kívánt objektumot kell átadnod.)

ArrayList

add(Object elem)
Hozzáadja az objektum paramétert a listához.

remove(int index)
Eltávolítja az index paraméter által jelzett helyen álló objektumot.

remove(Object elem)
Eltávolítja a megadott objektumot (amennyiben megtalálható a tömblistában).

contains(Object elem)
„True"-t ad vissza, ha a tömblistában van egyezés az objektum paraméterrel.

isEmpty()
„True"-t ad vissza, ha a tömblistában nincsenek elemek.

indexOf(Object elem)
Az objektum paraméter sorszámát (indexét) vagy –1-et ad vissza.

size()
A listában jelenleg található elemek számát adja vissza.

get(int index)
A jelenleg az index paraméter által jelzett helyen levő objektumot adja vissza.

Ez csak NÉHÁNY az ArrayList tagfüggvényei közül.

Néhány dolog, amit az ArrayList-tel csinálhatsz

1 Létrehozhatsz egyet

Most ne foglalkozz ezzel az új, csúcsos zárójeles <Tojas> jelölésse!. Csak annyit tesz, hogy „legyen ez egy Tojas objektumokból álló lista".

```
ArrayList<Egg> myList = new ArrayList<Egg>();
```

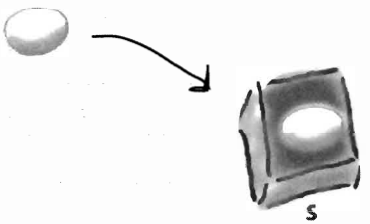
<Tojás>

Egy új ArrayList objektum jön létre a kupacban. Kicsi, mert üres.

2 Beletehetsz valamit

```
Egg s = new Egg();

myList.add(s);
```

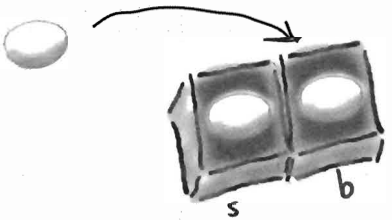


Az ArrayList most növeszt egy „dobozt", hogy tárolhassa a Tojas objektumot.

3 Beletehetsz egy másik dolgot

```
Egg b = new Egg();

myList.add(b);
```



Az ArrayList ismét nő egy kicsit, hogy beleférjen a második Tojas objektum is.

4 Kideríthetted, hány eleme van

```
int theSize = myList.size();
```

aMéret

Az ArrayList most 2 objektumot tárol, ezért a size() tagfüggvény a 2 értéket adja vissza

5 Kideríthetted, hogy van-e benne valami

```
boolean isIn = myList.contains(s);
```

Az ArrayList TARTALMAZZA az „s" által hivatkozott Tojas objektumot, ezért a contains() true-t ad vissza

6 Kideríthetted egy elemének a helyét (vagyis az elem sorszámát)

```
int idx = myList.indexOf(b);
```

Az ArrayList nulla alapú (vagyis az első sorszáma 0), és mivel a „b" által hivatkozott objektum a lista második eleme, az indexOf() 1-et ad vissza

7 Kideríthetted, hogy a lista üres-e

```
boolean empty = myList.isEmpty();
```

A lista határozottan NEM üres, ezért az isEmpty() false értéket ad vissza

8 Eltávolíthatsz belőle valamit

```
myList.remove(s);
```



Né' már! Összement!



Hegyezd ki a ceruzád!

Töltsd ki az alábbi táblázat hiányzó részeit a bal oldalon látható ArrayList-kód alapján, és írd be, hogy szerinted milyen utasításokra kellene cserélni a kódot, ha normál tömböt használnánk. Nem várjuk el, hogy minden abszolút helyes legyen, nyugodtan találgass a legjobb tudásod szerint.

ArrayList	normál tömb
`ArrayList<String> myList = new ArrayList<String>();`	String [] myList = new String[2];
`String a = new String("whoohoo");`	String a = new String("whoohoo");
`myList.add(a);`	
`String b = new String("Frog");`	String b = new String("Frog");
`myList.add(b);`	
`int theSize = myList.size();`	
`Object o = myList.get(1);`	
`myList.remove(1);`	
`boolean isIn = myList.contains(b);`	

Nincsenek
hülye kérdések

K: Az ArrayList király, de honnan tudhattam volna, hogy létezik?

V: A kérdés valójában ez: „Honnan *tudjam*, mi van az API-ban?". Márpedig ez a kulcsa annak, hogy sikeres Java-programozóvá válj, nem beszélve arról, hogy a lehető legtöbbet lustálkodhass, miközben mégis el tudsz készíteni egy programot. Meg fogsz lepődni, mennyit időt takaríthatsz meg annak köszönhetően, hogy valaki más már elvégezte a munka nehezét, és neked csak a szórakoztató résszel kell foglalkoznod.
De eltértünk a tárgytól… A válasz röviden az, hogy szánnod kell rá egy kis időt, hogy megtanuld, mi van a központi API-ban. A válasz bővebb kifejtését a fejezet végén találod, ahol azt is megtudod majd, *hogyan* ismerheted meg az API tartalmát.

K: Ez azért elég fontos kérdés. Nem csak azt kell tudnom, hogy a Java-könyvtárban van egy ArrayList osztály, hanem ami még fontosabb, azt is, hogy az ArrayList az, amire szükségem van! Hogyan jutok el a „csinálnom kell valamit" pontból az „így tudom megcsinálni" pontba az API segítségével?

V: Most beletrafáltál a lényegbe.

Mire végzel ezzel a könyvvel, jól fogod ismerni a Java nyelvet. Ezen kívül lényegében azt kell megtanulnod, hogy miként juthatsz el egy problémától a megoldásig, a lehető legkevesebb kód megírásával. Légy türelemmel még néhány oldal erejéig; a kérdéssel e fejezet végén már foglalkozni kezdünk.



Java – lehull a lepel

Heti interjúnk: ArrayList a tömbökről

Agyhullám: A tömbök tehát hasonlítanak a tömblistákhoz, igaz?

ArrayList: Álmodik a nyomor! ***Én*** *objektum* vagyok, köszönöm a kérdést.

Agyhullám: De ha nem tévedek, a tömbök is objektumok. A kupacban laknak az összes többi objektummal együtt.

ArrayList: Persze, a tömbök is a kupacba kerülnek – ***ó, jaj*** – , de egy tömb attól még csak szeretne ArrayList lenni. Csak egy ripacs. Az objektumoknak állapotuk *és* viselkedésük is van, nemdebár? Ezt tisztáztuk. Próbált már meghívni egy tagfüggvényt egy tömbre?

Agyhullám: Most, hogy említi, nem rémlik. De egyébként is, milyen tagfüggvényt hívnék meg? Engem csak az érdekel, hogy azokra a dolgokra hívhassak meg tagfüggvényeket, amelyeket *belet*ettem a tömbbe, nem magára a tömbre. A tömbformára akkor van szükségem, amikor beteszek vagy kiveszek valamit.

ArrayList: Valóban? Azt akarja mondani, hogy előfordult olyan, hogy ténylegesen *tö-rölt* valamit egy tömbből? Szent Kleofás, hol *képzik* magukat? A McJava's-ban?

Agyhullám: *Természetesen* vettem már ki dolgokat tömbökből. Azt mondom, hogy Kutya k = kutyaArray[1], és megkapom az 1-es sorszámon található Kutya objektumot.

ArrayList: Rendben, lassan mondom, hogy tudjon követni. Ilyenkor *nem* – ismétlem, *nem* – távolítja el azt a Kutyát a tömbből. Csak másolatot készít a *Kutyára mutató hivatkozásból*, és egy másik Kutya változóhoz rendeli azt.

Agyhullám: Á, értem, mire gondol. Nem, igazából tényleg nem töröltem a Kutya objektumot a tömbből; még mindig ott van. De azt hiszem, a hivatkozását null-ra állíthatom.

ArrayList: Én viszont egy első osztályú objektum vagyok: vannak tagfüggvényeim, és tudja, én *tényleg* képes vagyok olyasmikre, mint hogy eltávolítsam magamból egy Kutya hivatkozását, és ne csak null-ra állítsam. És még a méretemet is tudom változtatni. *Dinamikusan!* (Nézzen utána a szótárban.) Próbálja meg ezt egy *tömbbel*!

Agyhullám: Nem akartam ezt felhozni, de azt pletykálják, hogy maga csak egy agyonsztárolt, de kevéssé hatékony tömb. Hogy maga csak egy burkoló egy tömb számára, tagfüggvényekkel kiegészítve, hogy ne nekünk kelljen átméretező függvényeket és hasonlókat írni. És ha már itt tartunk: *maga még primitíveket se tud tárolni*! Ez komoly hiányosság, nem?

ArrayList: Nem hiszem el, hogy maga bevette ezt a maszlagot. Nem, én nem csak egy „kevéssé hatékony tömb" vagyok. Bevallom, néhány *rendkívül* ritka esetben egy tömb egy icuri-picurival – de tényleg csak egy *icuri-picurival* – gyorsabb lehet nálam. Ez a *jelentéktelen* sebességkülönbség azonban nem ad okot arra, hogy sutba dobjam az *erőmet*. Nézze, milyen *rugalmas* vagyok! Ami pedig a primitíveket illeti, *természetesen* beletehet egy primitívet egy ArrayList-be, feltéve hogy egy primitív burkolóosztályba csomagolja (erről bővebben a 10. fejezetben lesz szó). A Java 5.0 óta ráadásul ez a becsomagolás (és a kicsomagolás, amikor kiveszi a primitívet) önműködően történik. Rendben, *elismerem*, hogy ha *primitívekből* álló tömblistát használ, akkor a becsomagolás–kicsomagolás miatt valószínűleg jobban jár egy sima tömbbel, de akkor is…ki használ még *manapság* primitíveket?
Ó, hogy elszaladt az idő! *Lekésem a Pilates-edzésemet.* Folytassuk legközelebb!

Az ArrayList és a normál tömbök összehasonlítása

ArrayList	normál tömb
`ArrayList<String> myList = new ArrayList<String>();`	`String[] myList = new String[2];`
`String a = new String("whoohoo");`	`String a = new String("whoohoo");`
`myList.add(a);`	`myList[0] = a;`
`String b = new String("Frog");`	`String b = new String("Frog");`
`myList.add(b);`	`myList[1] = b;`
`int theSize = myList.size();`	`int theSize = myList.length;`
`Object o = myList.get(1);`	`String o = myList[1];`
`myList.remove(1);`	`myList[1] = null;`

`boolean isIn = myList.contains(b);`	`boolean isIn = false;`
	`for (String item : myList) {`
	`if (b.equals(item)) {`
	`isIn = true;`
	`break;`
	`}`
	`}`

Itt kezd nagyon különbözni!

Figyeld meg, hogy az ArrayList esetében egy ArrayList típusú objektummal dolgozol, így nyugodtan meghívhatod a régi jó tagfüggvényeket egy régi jó objektumra, a régi jó pont művelet segítségével.

A *tömbök* esetében a *különleges tömbutasításokat* (például myList[0] = foo) kell használnod, amiket sehol máshol nem alkalmazhatsz, csak a tömböknél. Bár a tömbök is objektumok, a saját kis világukban élnek, és nem hívhatsz meg rájuk tagfüggvényeket, csak az egyetlen példányváltozójukhoz, a *hosszukhoz* (*length*) férhetsz hozzá.

Az ArrayList és a normál tömbök összehasonlítása

1 A régi, egyszerű tömböknek ismerniük kell a méretüket, amikor létrehozod őket.

Az ArrayList esetében azonban csak létrehozol egy ArrayList típusú objektumot. Ez minden alkalommal így megy. A tömblistának soha nem kell tudnia, hogy milyen nagynak kell lennie, mivel úgy nő és zsugorodik, ahogy objektumokat adsz hozzá vagy veszel el belőle.

```
new String[ 2]
```

Méretre van szüksége.

```
new ArrayList<String>()
```

Nincs szükség méretre (bár adhatsz neki, ha akarsz).

2 Egy objektumot csak úgy tehetsz egy normál tömbbe, ha egy adott helyhez rendeled azt.

(Az adott hely egy sorszám, 0 és a tömb hossza mínusz egy között.)

```
myList[ 1] = b;
```

Meg kell adnod egy sorszámot.

Ha a sorszám a tömb határain kívülre esik (például ha a tömböt 2 mérettel vezetted be, most pedig a 3-as sorszámú helyhez próbálsz rendelni valamit), akkor a tömb futáskor felrobban.

Az ArrayList esetében ezzel szemben megadhatsz egy sorszámot az *add(egyInt, egyObjektum)* tagfüggvénnyel, vagy mondhatsz csak annyit, hogy *add(egyObjektum)*, és az ArrayList addig nő, amíg új dolgokat teszel bele.

```
myList.add(b);
```

Nincs sorszám.

3 A tömbök olyan utasításformát követnek, amit a Javában sehol máshol nem használsz.

A tömblisták ezzel szemben régi, jól bevált Java-objektumok, ezért nem kell hozzájuk különleges utasításforma.

```
myList[ 1]
```

A szögletes tömbzárójelek különleges utasításformáját csak a tömbökhöz kell használni.

4 Az ArrayList objektumok a Java 5.0-ban paraméterezettek.

Az imént említettük, hogy a tömbökkel ellentétben a tömblisták nem igényelnek különleges utasításformát. Valami különlegeset azonban mégiscsak használnak, ami a Java 5.0-ban (a Tigerben) került a nyelvbe: ***paraméterezett típusokat***.

```
ArrayList<String>
```

A csúcsos zárójelek között álló <String> egy „típusparaméter". Az ArrayList<String> jelentése egyszerűen annyi, hogy „karakterláncok listája", míg az ArrayList<Kutya> annyit tesz, hogy „Kutyák listája".

A Java 5.0 előtt nem volt mód a tömblistákba kerülő dolgok *típusának* megadására, így a fordító számára minden ArrayList csupán különféle objektumok gyűjteménye volt. Az <ideJönATípus> jelölésnek köszönhetően azonban most már bevezethetsz egy olyan tömblistát, amelyik tudja (és így korlátozhatja), hogy milyen típusú objektumokat tárolhat. A paraméterezett típusokat később részletesen is tárgyaljuk, ezért egyelőre ne töprengj túl sokat a csúcsos zárójeleken, amelyeket a tömblistáknál látsz. Elég, ha tudod, hogy ezek kényszerítik rá a fordítót, hogy csak egy bizonyos típusú objektumot (a csúcsos zárójelek között álló típust) engedjen meg a tömblistában.

Javítsd ki a DotCom kódját!

Emlékeztetőül álljon itt a hibás változat:

```
public class DotCom {

   int[] locationCells;
   int numOfHits = 0;

   public void setLocationCells(int[] locs) {
      locationCells = locs;
   }

   public String checkYourself(String stringGuess) {
      int guess = Integer.parseInt(stringGuess);
      String result = "miss";

      for (int cell : locationCells) {
         if (guess == cell) {

            result = "hit";
            numOfHits++;

            break;
         }
      } // ki a ciklusból

      if (numOfHits == locationCells.length) {
          result = "kill";
      }
      System.out.println(result);
         return result;
   } // a tagfüggvény lezárása
} // az osztály lezárása
```

Az új, nagyobb tudású változathoz átneveztük az osztályt DotCom-ra (SimpleDotCom-ról), de a kód ugyanaz, mint amit az előző fejezetben láttál.

Itt siklott ki a dolog. Minden tippet találatként számoltunk, akkor is, ha az adott cellát korábban már érte találat.

kódvázlat tesztkód valódi kód

Az új, továbbfejlesztett DotCom osztály

Most az ArrayList erejével!

```
import java.util.ArrayList;
```

Ezzel a sorral most ne foglalkozz; majd a fejezet végén beszélünk róla.

```
public class DotCom {

   private ArrayList<String> locationCells;
// private int numOfHits
// erre most nincs szükség
```

A String tömböt karakterláncokat tároló ArrayList-té alakítjuk.

```
   public void setLocationCells(ArrayList<String> loc) {
      locationCells = loc;
   }
```

Új, találóbb név az argumentumnak.

```
   public String checkYourself(String userInput) {

      String result = "miss";
```

A sorszám lekérdezésével kideritjük, hogy a felhasználó tippje benne van-e a tömblistában. Ha nincs, akkor az indexOf() –1-et ad vissza.

```
      int index = locationCells.indexOf(userInput);

      if (index >= 0) {
```

Ha a sorszám nulla vagy annál nagyobb, a felhasználó tippje biztosan a listában van, így törölhetjük.

```
         locationCells.remove(index);
```

```
         if (locationCells.isEmpty()) {
```

Ha a lista üres, akkor ez volt a végső csapás!

```
            result = "kill";
         } else {
            result = "hit";
         } // az if lezárása

      } // a külső if lezárása

      return result;
   } // a tagfüggvény lezárása
} // az osztály lezárása
```

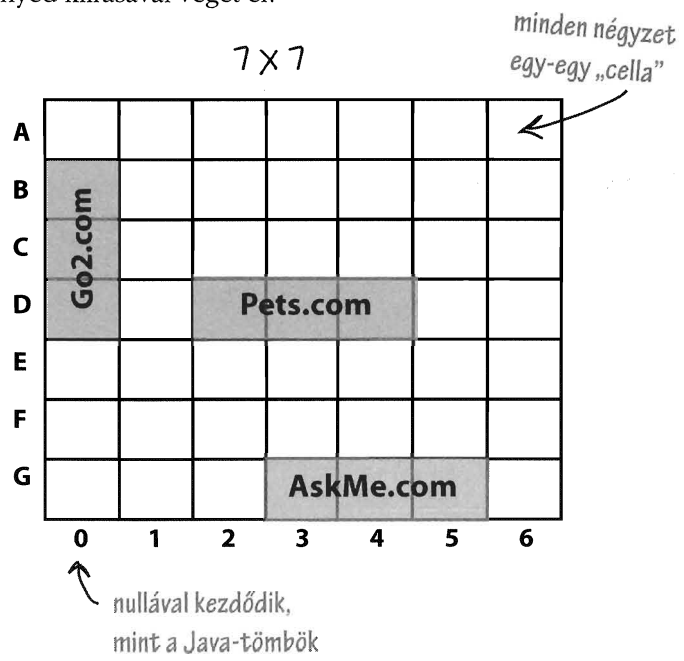
Építsd fel az IGAZI játékot – Sink a Dot Com

Eddig az „egyszerű” változaton dolgoztál, de most már ideje, hogy elkészítsd az igazit, amelyben egyetlen sor helyett egy rács található, és egyetlen DotCom helyett három.

A cél: Elsüllyeszteni a számítógép Dot Com-jait, a lehető legkevesebb lövéssel. A végén a teljesítményed alapján értékelést vagy szintbesorolást kapsz.

Előkészítés: Amikor elindítod a programot, a számítógép három Dot Com-ot helyez el egy **7 x 7-es virtuális rácson**. Ha ezzel elkészült, a játék kéri az első tippet.

A játék menete: Még nem tanultad meg, hogyan kell grafikus felületet építeni, ezért ez a változat parancssori utasításokkal működik. A számítógép felszólít, hogy írj be egy tippet (egy cella koordinátáit), amit így adhatsz meg: „A3”, „C5” stb. A lövés eredményét is a parancssorban láthatod, „Hit” (Talált), „Miss” (Nem talált) vagy „You sunk Pets.com” (Pets.com talált, süllyedt) formában (vagy bármelyik Dot Com cégnek legyen is szerencsés napja). Ha mind a három Dot Com-ot a 404-es süllyesztőbe küldted, a játék az elért eredményed kiírásával véget ér.



A Sink a Dot Com játékot fogod elkészíteni, egy 7 x 7-es ráccsal és három Dot Com céggel. Minden Dot Com három cellát foglal el.

részlet a játék üzeneteiből

```
File Edit Window Help Sell
%java DotComBust
Enter a guess  A3
miss
Enter a guess  B2
miss
Enter a guess  C4
miss
Enter a guess  D2
hit
Enter a guess  D3
hit
Enter a guess  D4
Ouch! You sunk Pets.com   : (
kill
Enter a guess  B4
miss
Enter a guess  G3
hit
Enter a guess  G4
hit
Enter a guess  G5
Ouch! You sunk AskMe.com   : (
```

Mit kell módosítanod?

A játék három osztályát kell módosítanod: a Dot Com-ok osztályát (amit most már nem SimpleDotCom-nak, hanem DotCom-nak hívunk), a játék osztályát (DotComBust) és a játék segédosztályát (amivel most nem foglalkozunk).

A DotCom osztály

- **Ehhez az osztályhoz hozzá kell adnod egy *name* (név) változót**
  a DotCom nevének („Pets.com", „Go2.com" stb.) tárolására, hogy minden DotCom kiírhassa a nevét, amikor elsüllyesztik (lásd a kimenet ablakát a következő oldalon).

B DotComBust osztály (a játék)

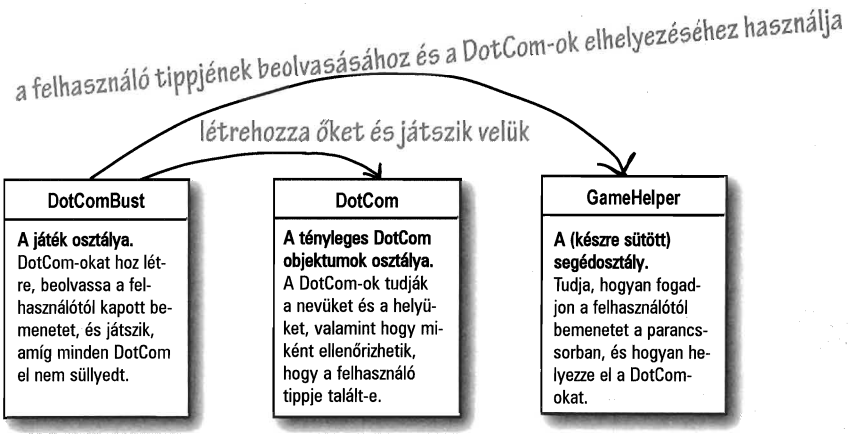
- **Egy helyett *három* DotCom-ot kell létrehoznod.**
- **Mindhárom DotCom-nak *nevet* kell adnod.**
  Egy beállító tagfüggvényt fogsz meghívni minden egyes DotCom-példányra, hogy a DotCom a name példányváltozójához rendelhesse a nevét.

DotComBust osztály (folytatás)

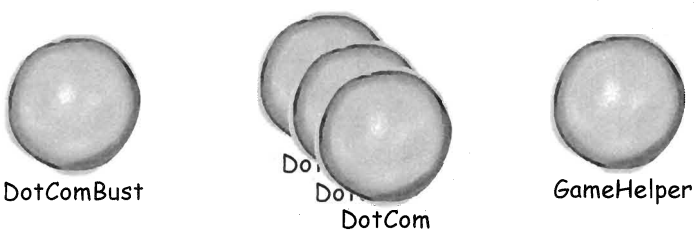
- **A Dot Com-okat nem egyetlen sorban, hanem egy rácson kell elhelyezned, és ezt mindhárom DotCom-mal el kell végezned.**
  Ez a lépés semmivel sem bonyolultabb, mint korábban, ha a DotCom-okat véletlenszerűen helyezed el. Mivel matekozni nem szeretnénk, a DotCom-ok helyét meghatározó algoritmust a (készre sütött) GameHelper segédosztályba tettük.
- **A felhasználó tippjeit nem csak egy, hanem mindhárom Dot Com helyzetével össze kell vetned.**
- **Addig kell folytatni a játékot** (vagyis elfogadni a felhasználó tippjeit, és összevetni azokat a fennmaradó DotCom-okkal), ***amíg el nem fogynak az élő DotCom-ok***.
- **Ki kell lépned a main tagfüggvényből.** Az egyszerű változatot végig a main-ben tartottuk, hogy…egyszerű legyen – az *igazi* játékban azonban nem szeretnénk ezt.

---

3 osztály:

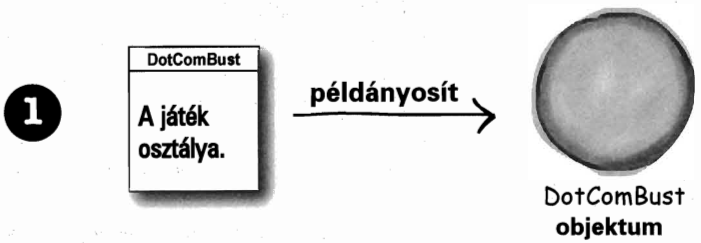


5 osztály:

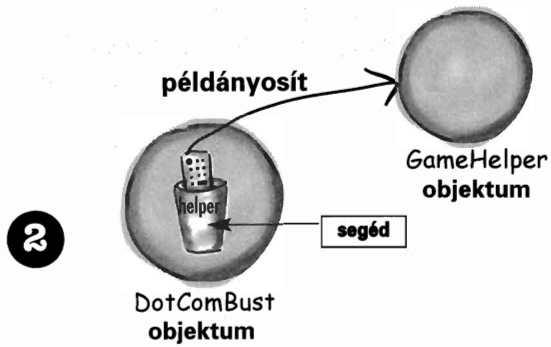


Továbbá 4 ArrayList: egy a DotComBust számára, és egy-egy a három DotCom objektumhoz.

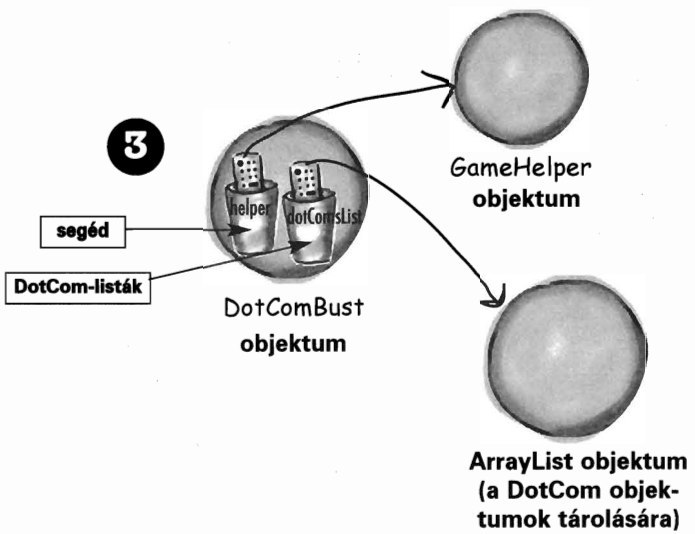
Mi mit (és mikor) csinál a DotComBust játékosztályban?



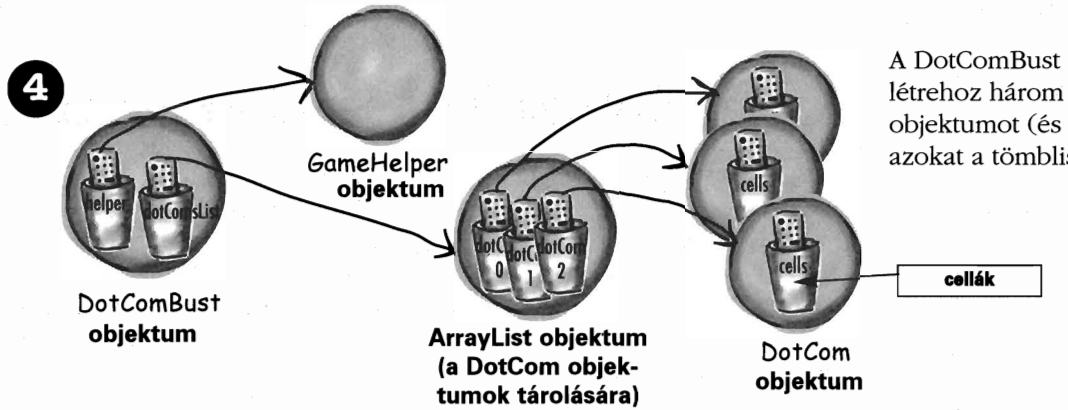
A DotComBust osztály main() tagfüggvénye létrehozza a DotComBust-objektumpéldányt, ami az egész játékot működteti.



A DotComBust objektum (a játék) létrehoz egy GameHelper-példányt, vagyis azt az objektumot, amelyik segít a játéknak elvégezni a munkáját.

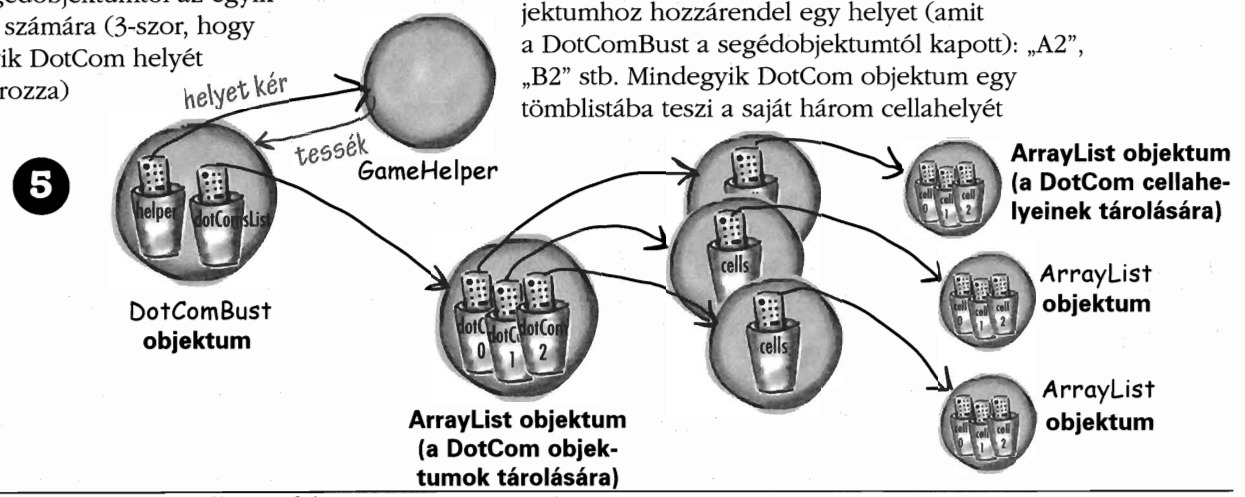


A DotComBust objektum létrehoz egy ArrayList példányt, amely a 3 DotCom objektumot tárolja majd.



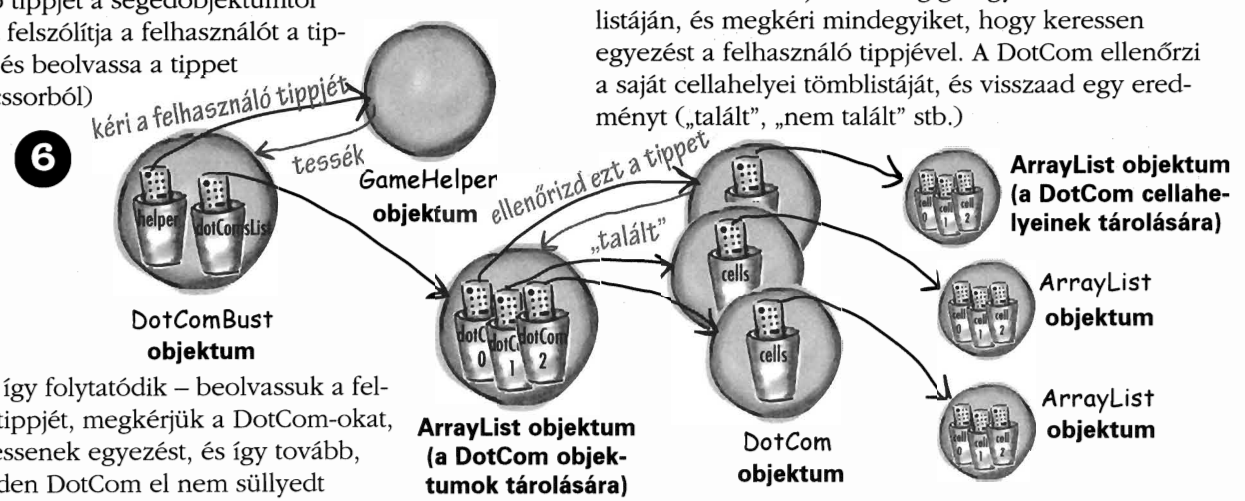
A DotComBust objektum helyet kér a segédobjektumtól az egyik DotCom számára (3-szor, hogy mindegyik DotCom helyét meghatározza)

A DotComBust objektum minden DotCom objektumhoz hozzárendel egy helyet (amit a DotComBust a segédobjektumtól kapott): „A2", „B2" stb. Mindegyik DotCom objektum egy tömblistába teszi a saját három cellahelyét

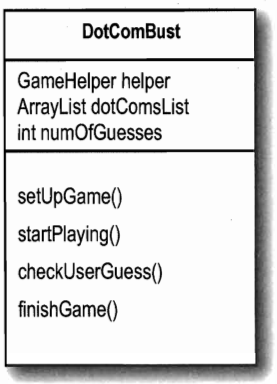


A DotComBust objektum elkéri a felhasználó tippjét a segédobjektumtól (a segéd felszólítja a felhasználót a tippelésre, és beolvassa a tippet a parancssorból)

A DotComBust objektum végigmegy a DotCom-ok listáján, és megkéri mindegyiket, hogy keressen egyezést a felhasználó tippjével. A DotCom ellenőrzi a saját cellahelyei tömblistáját, és visszaad egy eredményt („talált", „nem talált" stb.)



És a játék így folytatódik – beolvassuk a felhasználó tippjét, megkérjük a DotCom-okat, hogy keressenek egyezést, és így tovább, amíg minden DotCom el nem süllyedt



Az igazi DotComBust osztály kódvázlata

A DotComBust osztálynak három fő feladata van: a játék előkészítése, a játék lejátszása, amíg minden DotCom el nem süllyedt, illetve a játék befejezése. Bár ezt a három feladatot közvetlenül is leképezhettük volna három tagfüggvényre, úgy döntöttünk, hogy a középsőt (a játék lejátszását) *két* tagfüggvényre bontjuk, hogy kisebb kódszemcséket kapjunk. A kisebb tagfüggvények (vagyis amelyek kevesebb képességgel rendelkeznek) megkönnyítik a kód tesztelését, hibamentesítését és módosítását.

A változók bevezetése

BEVEZET és példányosít a ***GameHelper példányváltozó***, ***helper*** (segéd) néven.
BEVEZET és példányosít egy ***ArrayList*** a DotCom-ok listájának tárolására (kezdetben három van belőlük). A neve legyen ***dotComsList***.
BEVEZET egy ***int*** változó a felhasználó tippjei számának tárolására (hogy a játék végén kiírhassunk a játékos eredményét). A neve legyen ***numOfGuesses***, és állítsuk 0-ra.

A tagfüggvények bevezetése

BEVEZET ***setUpGame()*** tagfüggvény a DotCom objektumok létrehozására és előkészítésére egy névvel és egy hellyel. Rövid szabályismertető megjelenítése a felhasználó számára
BEVEZET ***startPlaying()*** tagfüggvény, amely tippeket kér a játékostól, és addig hívogatja a checkUserGuess() tagfüggvényt, amíg minden DotCom objektumot nem töröltünk a játékból.
BEVEZET ***checkUserGuess()*** tagfüggvény, amely végigmegy a fennmaradó DotCom objektumokon, és meghívja mindegyiknek a checkYourself() tagfüggvényét.
BEVEZET ***finishGame()*** tagfüggvény, amely kiírja a játékos eredményét annak alapján, hogy hány lövésre volt szüksége az összes DotCom objektum elsüllyesztéséhez.

A tagfüggvények megvalósítása

TAGFÜGGVÉNY: ***void setUpGame()***
// létrehoz három DotCom objektumot, és nevet ad nekik
LÉTREHOZ három DotCom objektum
BEÁLLÍT név minden DotCom számára
HOZZÁAD a DotCom-ok a ***dotComList***-hez (az ArrayList)
ISMÉTEL a ***dotComsList*** tömbben levő összes DotCom objektummal
MEGHÍV ***placeDotCom()*** tagfüggvény a segédobjektumra, hogy véletlenszerű helyet kapjon az adott DotCom számára (három cella egymás mellett vagy alatt egy 7 x 7-es rácsban)
BEÁLLÍT az egyes DotCom-ok helye a ***placeDotCom()*** hívás eredménye alapján
ISMÉTEL VÉGE
TAGFÜGGVÉNY VÉGE

kódvázlat tesztkód valódi kód

A tagfüggvények megvalósítása (folytatás):

TAGFÜGGVÉNY: ***void startPlaying()***

ISMÉTEL amíg van élő DotCom
KIOLVAS a felhasználó tippje a segéd ***getUserInput()*** tagfüggvényének meghívásával
KIÉRTÉKEL a felhasználó tippje a ***checkUserGuess()*** tagfüggvénnyel
ISMÉTEL VÉGE
TAGFÜGGVÉNY VÉGE

TAGFÜGGVÉNY: ***void checkUserGuess (String userGuess)***

// kideríti, hogy kapott-e találatot (vagy elsüllyedt-e) valamelyik DotCom
NÖVEL a felhasználó tippjeinek száma a ***numOfGuesses*** változóban
BEÁLLÍT a helyi ***result*** (eredmény) változó (egy ***String***) „miss" értékre, feltételezve, hogy a lövés nem talál
ISMÉTEL a ***dotComsList*** minden DotCom objektumára
KIÉRTÉKEL a felhasználó tippje a ***checkUserGuess()*** tagfüggvénnyel
BEÁLLÍT az eredményváltozó „hit" vagy „kill" értékre, szükség esetén
IF az eredmény „kill", **TÖRÖL** a DotCom a ***dotComsList***-ből
ISMÉTEL VÉGE
MEGJELENÍT a ***result*** értéke a felhasználó számára
TAGFÜGGVÉNY VÉGE

TAGFÜGGVÉNY: ***void finishGame()***

MEGJELENÍT általános „game over" üzenet, majd:
IF a felhasználói tippek száma kicsi,
MEGJELENÍT gratuláló üzenet
ELSE
MEGJELENÍT pocskondiázó üzenet
IF VÉGE
TAGFÜGGVÉNY VÉGE

Hegyezd ki a ceruzád!

Hogyan jutunk el a kódvázlattól a végleges kódig? A tesztkóddal kezdünk, majd tesztelünk, és apránként felépítjük a tagfüggvényeket. A könyvben nem fogjuk állandóan bemutatni a tesztkódot is, ezért most a te feladatod, hogy kitaláld, hogy mit kell tudnod ahhoz, hogy tesztelhesd ezeket a tagfüggvényeket. Melyik tagfüggvényt írnád meg és tesztelnéd először? Próbálj kódvázlatot írni néhány teszthez! Ennek a gyakorlatnak a megoldásaként kódvázlat helyett egy felsorolás is megteszi, de ha *igazi* (Java nyelvű) tesztkódot szeretnél írni, akkor oda kell tenned magad.

kódvázlat | tesztkód | valódi kód

```
import java.util.*;
public class DotComBust {

  private GameHelper helper = new GameHelper();                          (1)
  private ArrayList<DotCom> dotComsList = new ArrayList<DotCom>();
  private int numOfGuesses = 0;

  private void setUpGame() {
    // először létrehozzuk a dotcom-okat, és elhelyezzük őket
    DotCom one = new DotCom();
    one.setName("Pets.com");
    DotCom two = new DotCom();
    two.setName("eToys.com");
    DotCom three = new DotCom();                                         (2)
    three.setName("Go2.com");
    dotComsList.add(one);
    dotComsList.add(two);
    dotComsList.add(three);

    System.out.println("Your goal is to sink three dot coms.");          (3)
    System.out.println("Pets.com, eToys.com, Go2.com");
    System.out.println("Try to sink them all in the fewest number of guesses");

    for (DotCom dotComToSet : dotComsList) {                             (4)
      ArrayList<String> newLocation = helper.placeDotCom(3);             (5)
      dotComToSet.setLocationCells(newLocation);                         (6)
    } // a for ciklus lezárása
  } // a setUpGame tagfüggvény lezárása
  private void startPlaying() {
    while(!dotComsList.isEmpty()) {                                      (7)
      String userGuess = helper.getUserInput("Enter a guess");           (8)
      checkUserGuess(userGuess);                                         (9)
    } // a while lezárása
    finishGame();                                                        (10)
  } // a startPlaying tagfüggvény lezárása
```

Hegyezd ki a ceruzád!

Lásd el megjegyzésekkel a kódot!

Párosítsd össze a lap alján található megjegyzéseket a számozott kódrészekkel, és írd a megfelelő számot az egyes megjegyzések elé.

Minden megjegyzést csak egyszer használhatsz fel, és mindegyik megjegyzésre szükséged lesz.

bevezetjük és előkészítjük a szükséges változókat

beolvassuk a felhasználó tippjét

helyet kérünk a segédtől a DotCom számára

ezt a listában található mindegyik DotCom-mal megismételjük

rövid használati utasítást írunk ki a felhasználónak

meghívjuk a beállító tagfüggvényt az adott DotCom-ra, hogy hozzárendeljük azt a helyet, amelyet az imént kaptunk a segédtől

meghívjuk a saját checkUserGuess tagfüggvényünket

létrehozunk három DotCom objektumot, nevet adunk nekik, és belenyomjuk őket a tömblistába

amíg a DotCom-lista NEM üres

meghívjuk a saját finishGame tagfüggvényünket

– létrehozunk egy DotCom-objektumokból álló tömblistát (más szavakkal, egy listát, amely CSAK DotCom-objektumokat tárol, ahogy a DotCom[] is DotCom-objektumok tömbjét jelentené)

kódvázlat **tesztkód** valódi kód

```
private void checkUserGuess(String userGuess) {
    numOfGuesses++;                                          (11)
    String result  = "miss";                                 (12)
    for (DotCom dotComToTest : dotComsList) {                (13)
       result = dotComToTest.checkYourself(userGuess);       (14)
       if (result.equals("hit")) {
           break;                                            (15)
       }
       if (result.equals("kill")) {
           dotComsList.remove(dotComToTest);                 (16)
           break;
       }
    } // a for lezárása
    System.out.println(result);                              (17)
} // a tagfüggvény lezárása

private void finishGame() {
    System.out.println("All Dot Coms are dead! Your stock is now worthless.");
    if (numOfGuesses <= 18) {
       System.out.println("It only took you " + numOfGuesses + " guesses.");
       System.out.println(" You got out before your options sank.");
    } else {
       System.out.println("Took you long enough. "+ numOfGuesses + " guesses.");
       System.out.println("Fish are dancing with your options.");
    }
} // a tagfüggvény lezárása

public static void main (String[] args) {
    DotComBust game = new DotComBust();                      (19)
    game.setUpGame();                                        (20)
    game.startPlaying();                                     (21)
} // close method
} // a tagfüggvény lezárása
```

Semmiképpen NE lapozz, amíg el nem végezted a gyakorlatot! A mi megoldásunk a következő oldalon található. →

(18)

— ezt a listában található mindegyik DotCom-mal megismételjük

— ez a fickó kipurcant, ezért kivesszük a DotCom-listából, és kilépünk a ciklusból

— növeljük a felhasználó tippjeinek a számát

— idő előtt kilépünk a ciklusból, mert a többit nincs értelme megvizsgálni

— kiírunk egy üzenetet a felhasználónak arról, hogy hogyan teljesített

— feltételezzük, hogy „nem talált", hacsak nem kaptunk más eredményt

— utasítjuk a játékobjektumot, hogy indítsa el a fő játékciklust (ami folyamatosan tippeket kér a felhasználótól, és ellenőrzi azokat)

— kiírjuk az eredményt a felhasználónak

— utasítjuk a játékobjektumot, hogy készítse elő a játékot

— megkérjük a DotCom-ot, hogy ellenőrizze a felhasználó tippjét, hogy talált (vagy süllyesztett)-e

— létrehozzuk a játékobjektumot

kódvázlat | **tesztkód** | **valódi kód**

bevezetjük és előkészítjük a szükséges változókat

```
import java.util.*;
public class DotComBust {

    private GameHelper helper = new GameHelper();
    private ArrayList<DotCom> dotComsList = new ArrayList<DotCom>();
    private int numOfGuesses = 0;
```

létrehozunk egy DotCom-objektumokból álló tömblistát (más szavakkal, egy listát, amely CSAK DotCom-objektumokat tárol, ahogy a DotCom[] is DotCom-objektumok tömbjét jelentené)

```
    private void setUpGame() {
        // first make some dot coms and give them locations
        DotCom one = new DotCom();
        one.setName("Pets.com");
        DotCom two = new DotCom();
        two.setName("eToys.com");
        DotCom three = new DotCom();
        three.setName("Go2.com");
        dotComsList.add(one);
        dotComsList.add(two);
        dotComsList.add(three);
```

létrehozunk három DotCom objektumot, nevet adunk nekik, és belenyomjuk őket a tömblistába

```
        System.out.println("Your goal is to sink three dot coms.");
        System.out.println("Pets.com, eToys.com, Go2.com");
        System.out.println("Try to sink them all in the fewest number of guesses");
```

rövid használati utasítást írunk ki a felhasználónak

```
        for (DotCom dotComToSet : dotComsList) {
            ArrayList<String> newLocation = helper.placeDotCom(3);
            dotComToSet.setLocationCells(newLocation);
        } // close for loop
    } // close setUpgame method
```

ezt a listában található mindegyik DotCom-mal megismételjük

helyet (egy karakterláncokból álló tömblistát) kérünk a segédtől a DotCom számára

meghívjuk a beállító tagfüggvényt az adott DotCom-ra, hogy hozzárendeljük azt a helyet, amelyet az imént kaptunk a segédtől

```
    private void startPlaying() {
        while(!dotComsList.isEmpty()) {
            String userGuess = helper.getUserInput("Enter a guess");
            checkUserGuess(userGuess);
        } // close while
        finishGame();
    } // close startPlaying method
```

amíg a DotCom-lista NEM üres (a ! azt jelenti, hogy NEM; ez megegyezik azzal, hogy (dotComsList.isEmpty() == false)

beolvassuk a felhasználó tippjét

meghívjuk a saját checkUserGuess tagfüggvényünket

meghívjuk a saját finishGame tagfüggvényünket

kódvázlat **tesztkód** **valódi kód**

```
private void checkUserGuess(String userGuess) {
    numOfGuesses++;
    String result  = "miss";
    for (DotCom dotComToTest : dotComsList) {
       result = dotComToTest.checkYourself(userGuess);
       if (result.equals("hit")) {
             break;
       }
       if (result.equals("kill")) {
             dotComsList.remove(dotComToTest);
             break;
       }
    } // close for
    System.out.println(result);
} // close method
```

növeljük a felhasználó tippjeinek a számát

feltételezzük, hogy „nem talált", hacsak nem kaptunk más eredményt

ezt a listában található mindegyik DotCom-mal megismételjük

megkérjük a DotCom-ot, hogy ellenőrizze a felhasználó tippjét, hogy talált (vagy süllyesztett)-e

idő előtt kilépünk a ciklusból, mert a többit nincs értelme megvizsgálni

ez a fickó kipurcant, ezért kivesszük a DotCom-listából, és kilépünk a ciklusból

kiírjuk az eredményt a felhasználónak

kiírunk egy üzenetet a felhasználónak arról, hogy hogyan teljesített

```
private void finishGame() {
    System.out.println("All Dot Coms are dead! Your stock is now worthless.");
    if (numOfGuesses <= 18) {
       System.out.println("It only took you " + numOfGuesses + " guesses.");
       System.out.println(" You got out before your options sank.");
    } else {
       System.out.println("Took you long enough. "+ numOfGuesses + " guesses.");
       System.out.println("Fish are dancing with your options");
    }
} // close method
```

```
public static void main (String[] args) {
    DotComBust game = new DotComBust();
    game.setUpGame();
    game.startPlaying();
} // close method
```

létrehozzuk a játékobjektumot

utasítjuk a játékobjektumot, hogy készítse elő a játékot

utasítjuk a játékobjektumot, hogy indítsa el a fő játékciklust (ami folyamatosan tippeket kér a felhasználótól, és ellenőrzi azokat)

A DotCom osztály végleges változata

```
import java.util.*;

public class DotCom {
   private ArrayList<String> locationCells;
   private String name;

   public void setLocationCells(ArrayList<String> loc) {
      locationCells = loc;
   }

      public void setName(String n) {
      name = n;
  }

   public String checkYourself(String userInput) {
      String result = "miss";
      int index = locationCells.indexOf(userInput);
      if (index >= 0) {
          locationCells.remove(index);

         if (locationCells.isEmpty()) {
             result = "kill";
             System.out.println("Ouch! You sunk " + name + "   : ( ");
         } else {
             result = "hit";
         }  // close if
      } // close if
       return result;

   } // close method
} // close class
```

A DotCom példányváltozói:
– a cellahelyek tömblistája
– a DotCom neve

Beállító tagfüggvény, amely frissíti a DotCom helyét (a véletlenszerű helyet a GameHelper osztály placeDotCom() tagfüggvénye biztosítja)

Az ArrayList indexOf() tagfüggvénye működés közben. Ha a felhasználó tippje megtalálható a tömblistában, az indexOf() annak a listában elfoglalt helyét fogja visszaadni. Ha nem, akkor –1-et.

Bejegyzés törlése az ArrayList remove() tagfüggvényével

Az isEmpty() tagfüggvénnyel ellenőrizzük, hogy mindent cellahelyet sikerült-e eltalálni.

Értesítjük a felhasználót, ha egy DotCom elsüllyedt.

A visszaadott érték „miss" (nem talált), „hit" (talált) vagy „kill" (süllyedt) lehet.

Néhány hasznos logikai kifejezés

Eddig amikor logikai kifejezéseket használtunk a ciklusokban vagy `if` vizsgálatokban, azok meglehetősen egyszerűek voltak. Az elkövetkező „készre sütött" kódokban azonban összetettebb logikai kifejezéseket láthatsz majd, és bár tudjuk, hogy nem fogsz lesni, úgy gondoltuk, hogy ez remek alkalom arra, hogy megmutassuk, hogyan hozhatod lendületbe a kifejezéseket.

„ÉS" és „VAGY" műveletek (&&, ||)

Tegyük fel, hogy egy chooseCamera() – „valasztKamera()" – nevű tagfüggvényt írsz, amelyben egy csomó szabály vezérli a kamera kiválasztását. Választhatsz mondjuk 50 és 1000 dollár közötti árú kamerákat, de előfordulhat, hogy ennél pontosabban szeretnéd szabályozni az ártartományt. Valami ilyesmit szeretnél mondani:
„Ha az ár (price) a 300 *és* 400 dollár közötti *tartományba* esik, akkor válaszd az X-et."

```
if (price >= 300 && price < 400) {
    camera = "X";
}
```

Tegyük fel, hogy a kód egy része a tíz elérhető kameratípus (brand) közül csak *néhányra* vonatkozik:

```
if (brand.equals("A") || brand.equals("B") ) {
    // itt kizárólag az A vagy a B típussal
    // csinálunk valamit
}
```

A logikai kifejezések igen hosszúak és összetettek is lehetnek:

```
if ((zoomType.equals("optical") &&
    (zoomDegree >= 3 && zoomDegree <= 8)) ||
    (zoomType.equals("digital") &&
    (zoomDegree >= 5 && zoomDegree <= 12))) {
  // a megfelelő közelítő-távolító (zoom) művelet
}
```

Ha érdekelnek a technikai részletek, felmerülhet benned a kérdés, hogy mi a fenti műveletek *kiértékelési sorrendje* (precedence). Mi azt javasoljuk, hogy ahelyett, hogy szaktekintéllyé képeznéd magad az elsőbbség fennkölt világában, ***használj zárójelezést***, hogy egyértelművé tedd a kódjaidat.

Nem egyenlő (!= és !)

Tegyük fel, hogy egy ehhez hasonló programlogikát szeretnél: „egy bizonyos dolog a tíz elérhető kameramodellből *egy kivételével mindegyikre igaz*":

```
if (model != 2000) {
    // itt olyasmi áll, ami a 2000-es
    // modellre nem vonatkozik
}
```

Ha pedig objektumokat karakterláncként szeretnél összehasonlítani…

```
if (!brand.equals("X")) {
    // itt olyasmi áll, ami
    // az X típusra nem vonatkozik
}
```

Rövidzáras műveletek (&&, ||)

Az eddig vizsgált műveletek – az && és a || – úgynevezett ***rövidzáras*** (short circuit) műveletek. Az && esetében a kifejezés csak akkor lesz igaz, ha az && *mindkét* oldala igaz. Ha tehát a JVM azt látja, hogy egy &&-kifejezés bal oldala hamis, azonnal megáll. A jobb oldalt már pillantásra sem méltatja. Ehhez hasonlóan, a || esetében a kifejezés akkor igaz, ha *valamelyik* oldal igaz, tehát ha a JVM azt látja, hogy a bal oldal igaz, az egész kifejezést igaznak veszi, és a jobb oldalt már nem vizsgálja meg. Miért nagyszerű ez? Tegyük fel, hogy van egy hivatkozó változód, de nem tudod biztosan, hogy hozzá lett-e már rendelve egy objektumhoz. Ha nem, és egy tagfüggvényt ezzel az üres (objektumhoz nem rendelt) hivatkozó változóval próbálsz meghívni, NullPointerException kivételt kapsz. Próbáld inkább ezt:

```
if (refVar != null &&
      refVar.isValidType() ) {
    // "érvényes típus esetén
    // végrehajtandó" utasítások
}
```

Nem rövidzáras műveletek (&, |)

Az & és | műveletek logikai kifejezésekben használva ugyanúgy működnek, mint az && és a ||, azzal a különbséggel, hogy a JVM-et arra kényszerítik, hogy *mindig* megvizsgálja a kifejezés *mindkét* oldalát. Az & és a | jellemzően más környezetben, bitműveletekhez használatos.



Készre sütött kód

Ez a játék segédosztálya. A felhasználótól bemenetet fogadó tagfüggvény (amely bekéri a tippet a játékostól, és beolvassa azt a parancssorból) mellett a segéd Fontos Szolgáltatása a DotCom-ok cellahelyeinek létrehozása. A helyedben mi lassan hátrálnánk ettől a kódtól, és csak annyit tennénk vele, hogy begépeljük és lefordítjuk. Megpróbáltuk a lehető legrövidebbre összenyomni, hogy ne kelljen olyan sokat gépelned, de ez sajnos azzal járt, hogy a kód nem lett túl könnyen olvasható. Ne feledd azonban, hogy *enélkül* az osztály nélkül nem fogod tudni lefordítani a DotComBust játékosztályt.

```
import java.io.*;
import java.util.*;

public class GameHelper {

  private static final String alphabet = "abcdefg";
  private int gridLength = 7;
  private int gridSize = 49;
  private int [] grid = new int[ gridSize] ;
  private int comCount = 0;

  public String getUserInput(String prompt) {
     String inputLine = null;
     System.out.print(prompt + "  ");
     try {
       BufferedReader is = new BufferedReader(
        new InputStreamReader(System.in));
       inputLine = is.readLine();
       if (inputLine.length() == 0 )  return null;
     } catch (IOException e) {
       System.out.println("IOException: " + e);
     }
     return inputLine.toLowerCase();
  }

 public ArrayList<String> placeDotCom(int comSize) {
    ArrayList<String> alphaCells = new ArrayList<String>();
    String [] alphacoords = new String [ comSize] ;   // olyan koordinátákat tárol, mint az "f6"
    String temp = null;                               // ideiglenes karakterlánc összefűzéshez
    int [] coords = new int[ comSize] ;               // az aktuális tipp koordinátái
    int attempts = 0;                                 // a tippszámláló aktuális értéke
    boolean success = false;                          // jelző = sikerült jó helyet találni?
    int location = 0;                                 // az aktuális kezdőcella

    comCount++;                                       // az n-edik elhelyezendő dotcom
    int incr = 1;                                     // vízszintes lépésköz beállítása
    if ((comCount % 2) == 1) {                        // ha páratlan dotcom (elhelyezés függőlegesen)
      incr = gridLength;                              // függőleges lépésköz beállítása
    }

    while ( !success & attempts++ < 200 ) {           // a fő keresőciklus (32)
       location = (int) (Math.random() * gridSize);   // véletlenszerű kezdőpont meghatározása
        //System.out.print(" try " + location);
       int x = 0;                                     // az elhelyezendő dotcom n-edik cellahelye
        success = true;                               // sikert feltételezünk
        while (success && x < comSize) {              // szomszédos szabad helyeket keres
          if (grid[ location] == 0) {                 // ha még nincs használatban
```

Megjegyzés: további pontokért megpróbálhatod, hogy „kiveszed a megjegyzésből" a placeDotCom() tagfüggvény System.out.print(ln) sorait, csak hogy lásd, mit csinálnak. Ezek a kiíró utasítások lehetővé teszik, hogy „csalj", mivel megadják neked a DotCom-ok helyét, de a tesztelésben hasznosak.



Készre sütött kód

A GameHelper osztály kódja (folytatás)

```
            coords[ x++] = location;                          // hely mentése
            location += incr;                                 // próbálkozás a "következő" cellával
            if (location >= gridSize){                        // ha túlment a határokon - "alul"
              success = false;                                // kudarc
            }
            if (x>0 && (location % gridLength == 0)) {        // ha túlment a határokon
              success = false;                                // - "a jobb szélen"
            }                                                 // kudarc
          } else {
            // System.out.print(" used " + location);        // már használatban levő
            success = false;                                  // helyet találtunk
          }                                                   // kudarc
        }
      }                                                       // while vége

      int x = 0;                                              // a cellahely alfakoordinátákká
      int row = 0;                                            // alakítása
      int column = 0;
      // System.out.println("\n");
      while (x < comSize) {
        grid[ coords[ x]] = 1;                                // a fő rácspontok megjelölése
        row = (int) (coords[ x] / gridLength);                // "használt"-ként
        column = coords[ x] % gridLength;                     // a sorérték kiolvasása
        temp = String.valueOf(alphabet.charAt(column));       // a számszerű oszlopérték kiolvasása
                                                              // átalakítás alfává
        alphaCells.add(temp.concat(Integer.toString(row)));
        x++;
        // System.out.print("  coord "+x+" = " + alphaCells.get(x-1));
      }

      // System.out.println("\n");

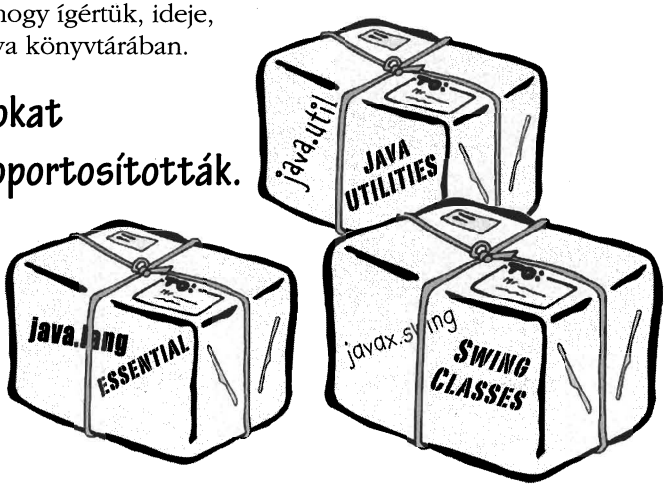
      return alphaCells;
    }
}
```

Ez az utasítás árulja el neked pontosan, hogy hol van a DotCom.

A Könyvtár (a Java API) használata

Az ArrayList segítségének köszönhetően átverekedted magad a DotComBust játékosztályon. Most pedig, ahogy ígértük, ideje, hogy megtanuld, hogyan igazodhatsz el a Java könyvtárában.

A Java API-ban az osztályokat csomagokba (package) csoportosították.



Ha használni szeretnéd az API valamelyik osztályát, tudnod kell, melyik csomagban található.

A Java könyvtárában minden osztály egy csomaghoz tartozik. A csomagoknak olyan nevük van, mint a **javax.swing** (ez a csomag tárolja a Swing GUI osztályainak egy részét; a Swingről hamarosan bővebben is beszélünk). Az ArrayList a **java.util** csomagban található, amely – minő meglepetés – egy halom *segédosztályt* (utility class) tartalmaz. A csomagokról bővebben tanulsz majd a 16. fejezetben; többek között megtudod, hogyan tehetsz *saját* osztályokat a *saját* csomagjaidba. Egyelőre azonban csak *használni* fogjuk a Java néhány beépített osztályát.

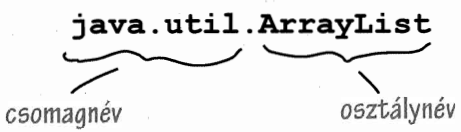
Az API osztályainak használata a saját kódjaidban egyszerű. Minden osztályt úgy kell kezelned, mintha magad írtad volna, és magad fordítottad volna le, és csak ülne ott, arra várva, hogy felhasználd.

Az egyetlen fontos különbséget az jelenti, hogy valahol a kódban meg kell adnod a használni kívánt könyvtárosztály *teljes* nevét, vagyis a csomag *és* az osztály nevét.

Még ha nem is tudtál róla, ***már használtál csomagból származó osztályokat***: a System (System.out.println), String és Math (Math.random()) egyaránt a **java.lang** csomag része.

Ha egy osztályt használni szeretnél a kódodban, tudnod kell a teljes nevét*.

Az ArrayList-nek nem ArrayList a *teljes* neve, mint ahogy a „Kathy" sem teljes név (nem úgy, mint Madonna vagy Cher művészneve, de ebbe ne menjünk bele). Az ArrayList teljes neve valójában ez:



Meg kell mondanod a Javának, hogy melyik ArrayList-et akarod használni. Erre két lehetőség kínálkozik:

A IMPORT

Elhelyezel egy betöltő utasítást (import) a forráskódot tartalmazó fájl elején:

```
import java.util.ArrayList;
public class MyClass {... }
```

VAGY

B TYPE

A kódban mindenhol beírod a teljes nevet, mégpedig minden alkalommal, amikor használod, *bárhol* is legyen.

Amikor bevezeted vagy példányosítod az osztályt:

```
java.util.ArrayList<Dog> list = new java.util.ArrayList<Dog>();
```

Amikor argumentum típusaként használod:

```
public void go(java.util.ArrayList<Dog> list) { }
```

Amikor visszatérési típusként használod:

```
public java.util.ArrayList<Dog> foo() {...}
```

*Hacsak az osztály nem a java.lang csomagban található.

Nincsenek

hülye kérdések

K: Miért kell teljes nevet használni? A csomagokra csak ezért van szükség?

V: A csomagok három okból kifolyólag fontosak. Először is, segítik a projektek és könyvtárak általános rendszerezését. Egyetlen ijesztően hatalmas osztályhalom helyett hasonló szolgáltatásokat nyújtó (grafikus felületi, adatszerkezeti, adatbázis-műveleti stb.) osztályokból álló csomagokat kapunk.
Másodszor, a csomagok névtereket biztosítanak, amelyek megakadályozzák az ütközéseket, ha történetesen neked és még vagy egy tucat programozó kollégádnak a cégnél van egy ugyanolyan nevű osztályotok. Ha egy osztálynak mondjuk a Set nevet adtad, és valaki másnak (beleértve a Java API-t is) szintén van egy Set nevű osztálya, akkor valahogy meg kell mondanod a JVM-nek, hogy *melyik* Set osztályt szeretnéd használatba venni.
Harmadszor, a csomagok bizonyos fokú védelmet nyújtanak, mivel a segítségükkel úgy korlátozhatod a hozzáférést az általad írt kódhoz, hogy csak a vele azonos csomagban található egyéb osztályok érhessék el. Erről bővebben a 16. fejezetben lesz szó.

K: Térjünk vissza egy kicsit a névütközéshez. Hogyan akadályozza ezt meg a teljes név? Mi akadályozza meg, hogy két ember ugyanazt a csomagnevet adja egy osztálynak?

V: A Java elnevezési rendszerének köszönhetően ez rendszerint nem történik meg, amennyiben a fejlesztők tartják magukat hozzá. Erre még bővebben visszatérünk a 16. fejezetben.

Honnan jön az „x"? (avagy, Mit jelent, ha egy csomag neve úgy kezdődik, hogy javax?)



A Java első és második kiadásában (1.02 és 1.1) a Java részét képező (más szavakkal, az alapkönyvtárban – standard library – helyet kapó) minden osztályt olyan csomagokba tettek, amelyeknek a neve a ***java*** szóval kezdődött. Ott volt természetesen a ***java.lang*** – az a csomag, amelyet nem kell külön betöltened –, aztán a ***java.net***, a ***java.io***, a ***java.util*** (bár az ArrayList akkoriban még nem létezett), és még pár további csomag, köztük a ***java.awt***, amely a grafikus felülettel kapcsolatos osztályokat tárolta.

A láthatáron azonban már ott várakozott néhány újabb csomag, amelyeket nem foglaltak bele az alapkönyvtárba. Ezeket a csomagokat ***bővítményként*** (extension) ismerték, és két fajtájuk létezett: a *szabványos* (standard) és a *nem szabványos* (not standard) bővítmények. Szabványos bővítménynek azokat tekintették, amelyeket a Sun cég hivatalosnak ismert el, szemben a kísérleti jellegű, próbára bocsátott vagy bétacsomagokkal, amelyekből aztán esetleg nem is készült végleges változat.

A megállapodás szerint minden szabványos bővítmény neve a szokványos ***java*** csomagnév-előtaggal, valamint ahhoz hozzáfűzve az „x" jelöléssel kezdődött. Minden szabványos bővítménynek a Swing könyvtár volt az őse, amelyben számos csomag helyet kapott, és mindegyiknek a neve úgy kezdődött, hogy ***javax.swing***.

A szabványos bővítmények azonban esélyesek arra, hogy első osztályú, a Javába beépített könyvtárcsomagok váljanak belőlük. Pontosan ez történt a Swinggel is az 1.2-es változattól kezdve (a Javának ez a változata lett az első, amelyet „Java 2"-nek hívtak).

„Király!" – gondolta mindenki (minket is beleértve) – „Most már mindenki, aki rendelkezik a Javával, hozzájut a Swing osztályaihoz, így nem nekünk kell kitalálnunk, hogy miként telepíttessük őket a végfelhasználóval."

A felszín alatt azonban veszély leselkedett: amikor egy csomagot előléptetnek, akkor a nevének TERMÉSZETESEN a ***java***, és nem a ***javax*** szóval kell kezdődnie. Mindenki TUDJA, hogy az alapkönyvtár csomagjainak a nevében nincs „x", csak a bővítményekében. Így aztán mielőtt az 1.2-es változatot véglegesítették volna (és ezt úgy értjük, hogy tényleg az utolsó pillanatban) a Sun megváltoztatta a csomagneveket, és (más módosítások mellett) törölte az „x"-et. Könyveket nyomtattak és dobtak piacra, amelyek már az új nevükön hivatkoztak a Swing-kódokra. Az elnevezési rendszer ép maradt, és a Java világa megmenekült.

Eltekintve attól a körülbelül 20 000, haját tépve üvöltő fejlesztőtől, aki ráébredt, hogy ez az egyszerű névváltoztatás szörnyű következményekkel jár: az összes olyan kódjukat, ami a Swingre támaszkodik, meg kell változtatniuk! Katasztrófa! Gondolj csak a rengeteg import ***javax*** kezdetű utasításra…

Így aztán a legvégső pillanatban, amikor a remény lángja már alig pislákolt, a kétségbeesett fejlesztők meggyőzték a Sunt, hogy „a francba a szabályokkal, meg kell menteni a kódokat!". A többi már történelem. Amikor tehát egy olyan csomaggal találkozol a könyvtárban, amelyiknek a neve úgy kezdődik, hogy ***javax***, tudd, hogy valaha bővítményként látta meg a napvilágot, csak aztán előléptették.

Újratöltve

- Az **ArrayList** (tömblista) a Java API egyik osztálya.
- Ha valamit bele szeretnél tenni egy ArrayList-be, az **add()** tagfüggvényt használhatod.
- Ha el szeretnél távolítani valamit egy ArrayList-ből, a **remove()** tagfüggvényt használhatod.
- Ha ki szeretnéd deríteni, hogy valami benne van-e egy ArrayList-ben, az **indexOf()** tagfüggvényt használhatod.
- Ha ki szeretnéd deríteni, hogy egy adott ArrayList üres-e, az **isEmpty()** tagfüggvényt használhatod.
- Egy ArrayList méretét (az elemeinek a számát) a **size()** ***tagfüggvénnyel*** kaphatod meg.
- Ha egy régi, szokványos tömb hosszára (az elemeinek a számára) vagy kíváncsi, a **length** ***változót*** kell kiolvasnod.
- A tömblisták ***dinamikusan átméreteződnek*** a szükséges méretre: bővülnek, ha objektumokat adsz hozzájuk, és **összezsugorodnak**, ha objektumokat távolítasz el belőlük.
- A tömbök típusát egy **típusparaméterrel** vezetheted be, ami egy csúcsos zárójelek között álló típusnév. Az ArrayList<Button> például azt jelenti, hogy a tömblista csak Button típusú (illetve – ahogy a következő néhány fejezetben megtanulod – a Button alosztályaiba tartozó) objektumokat lesz képes tárolni.
- Bár a tömblisták objektumokat, és nem primitíveket tárolnak, a fordítóprogram önműködően beburkolja (wrap) a primitíveket egy objektumba – illetve kivételkor kicsomagolja (unwrap) azokat –, és ezt az objektumot teszi a tömblistába a primitív helyett. (Erről a könyv későbbi részében még lesz szó.)
- Az osztályokat csomagokba csoportosították.
- Minden osztálynak van teljes neve, amely a csomagnévből, valamint az osztálynévből áll. Az ArrayList osztály teljes neve valójában java.util.ArrayList.
- Ha nem a java.lang, hanem egy másik csomag valamelyik osztályát szeretnéd használni, a Javának az osztály teljes nevét kell megadnod.
- Ha egy osztályt használatba szeretnél venni, vagy a forráskód elején kell betöltened egy import utasítással, vagy mindenhol meg kell adnod a teljes nevét a kódban, ahol felhasználod.

Nincsenek

hülye kérdések

K: Az `import` növeli az osztály méretét? A fordító ténylegesen belefordítja a betöltött osztályt vagy csomagot az *én* kódomba?

V: Netán egy C-programozóhoz van szerencsénk? Az `import` nem ugyanaz, mint az `include`. Tehát a válasz nem és nem. Mondd utánunk: „az `import` utasítás megment attól, hogy gépelnem kelljen". Ennyi az egész. Nem kell aggódnod amiatt, hogy a kódod felfúvódik, vagy lassabb lesz a túl sok betöltött osztálytól. Az `import` utasítás csak arra szolgál, hogy a Javának megmondd *egy osztály teljes nevét.*

K: Jó, de akkor miért nem kellett soha betöltenem a String vagy a System osztályt?

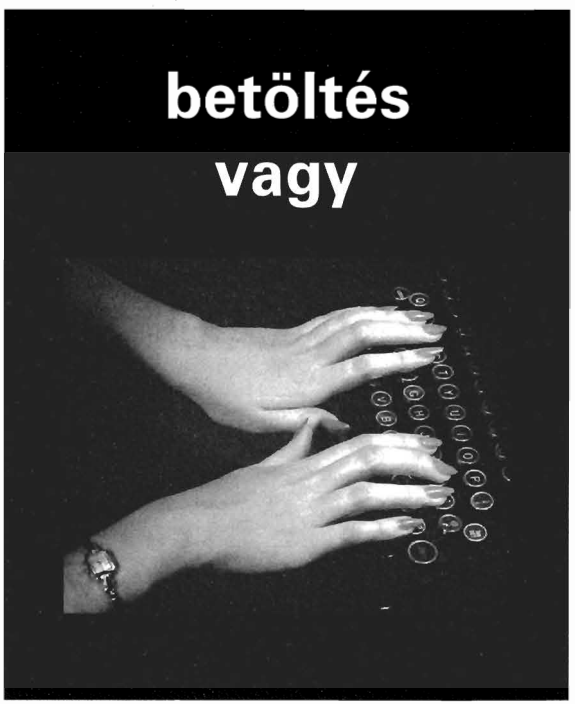
V: Emlékezz vissza, hogy a java.lang csomagot „ingyen előre betöltve" kapod. Mivel a java.lang csomagban található osztályok létfontosságúak, ezek esetében nem kell használnod a teljes nevet. Csak egy java.lang.String és egyetlen java.lang.System osztály létezik, a Java pedig nagyon jól tudja, hol találja őket.

K: A saját osztályaimat is csomagokba kell tennem? És ha igen, hogyan? Egyáltalán *megtehetem*?

V: A valóságban (amit jobb, ha elkerülsz) igen, az osztályaidat csomagokba *akarod* majd tenni. Erről részletesen a 16. fejezetben lesz szó. Egyelőre azonban a példakódjainkat nem tesszük csomagokba.



Még egyszer, arra a valószínűtlen esetre, ha még nem rögzült volna:



„Jó tudni, hogy van egy ArrayList osztály a java.util csomagban. De ezt magamtól hogy találtam volna ki?"
- Julia (31), kézmodell

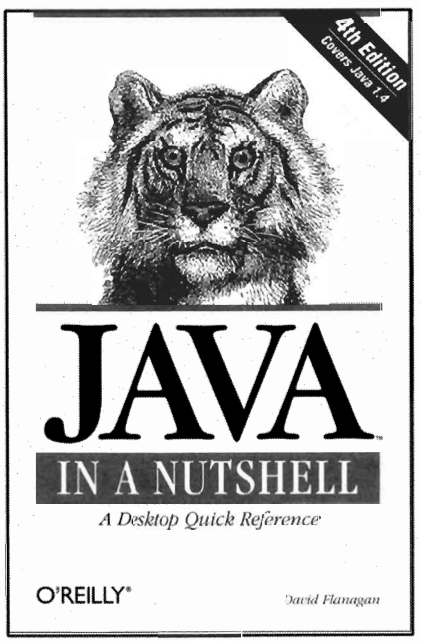


Hogyan játszhatsz az API-val?

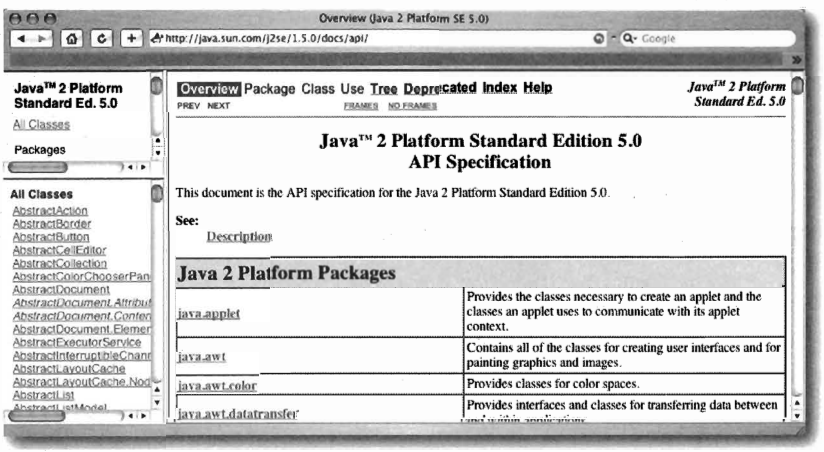
Két dologra keresed a választ:

1. **Milyen osztályok találhatók a könyvtárban?**
2. **Ha megtaláltál egy osztályt, honnan tudhatod, hogy mire képes?**

1 Üss fel egy Java-könyvet.



2 Nézd meg az API HTML-leírásában.



1 Üss fel egy Java-könyvet.



Egy referenciakönyv átlapozása a legjobb módja annak, hogy kideríts, mi van a Java könyvtárában. Csupán lapozgatással is könnyen rábukkanhatsz egy osztályra, amely hasznosnak tűnik.

java.util.Currency

Returned By: java.text.DecimalFormat.getCurrency(), java.text.DecimalFormatSymbols.getCurrency(), java.text.NumberFormat.getCurrency(), Currency.getInstance()

osztálynév →

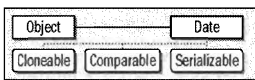
Date **Java 1.0**

csomagnév →

java.util ***cloneable serializable comparable***

az osztály leírása →

This class represents dates and times and lets you work with them in a system-independent way. You can create a Date by specifying the number of milliseconds from the epoch (midnight GMT, January 1st, 1970) or the year, month, date, and, optionally, the hour, minute, and second. Years are specified as the number of years since 1900. If you call the Date constructor with no arguments, the Date is initialized to the current time and date. The instance methods of the class allow you to get and set the various date and time fields, to compare dates and times, and to convert dates to and from string representations. As of Java 1.1, many of the date methods have been deprecated in favor of the methods of the Calendar class.




```
public class Date implements Cloneable, Comparable, Serializable {
// Public Constructors
    public Date();
    public Date(long date);
#   public Date(String s);
#   public Date(int year, int month, int date);
#   public Date(int year, int month, int date, int hrs, int min);
#   public Date(int year, int month, int date, int hrs, int min, int sec);
// Property Accessor Methods (by property name)
    public long getTime();
    public void setTime(long time);
// Public Instance Methods
    public boolean after(java.util.Date when);
    public boolean before(java.util.Date when);
1.2 public int compareTo(java.util.Date anotherDate);
// Methods Implementing Comparable
1.2 public int compareTo(Object o);
// Public Methods Overriding Object
1.2 public Object clone();
    public boolean equals(Object obj);
    public int hashCode();
    public String toString();
// Deprecated Public Methods
#   public int getDate();
#   public int getDay();
#   public int getHours();
#   public int getMinutes();
#   public int getMonth();
#   public int getSeconds();
#   public int getTimezoneOffset();
#   public int getYear();
#   public static long parse(String s);
#   public void setDate(int date);
#   public void setHours(int hours);
#   public void setMinutes(int minutes);
#   public void setMonth(int month);
```

tagfüggvények (és egyebek, amikről később lesz szó)

2 Nézd meg az API HTML-leírásában.

A Javához egy csodálatos internetes dokumentáció is tartozik, amit furcsa módon szintén Java API-nak hívnak. A leírások a Java 5 Standard Edition Documentation nevű (bár attól függően, hogy a hét melyik napján nézed, a Sun esetleg „Java 2 Standard Edition 5.0" néven hivatkozik rá) nagyobb gyűjtemény részét képezik, és külön kell letöltened őket, tehát nem zsugorfóliázzák össze a Java 5 letölthető csomagjával. Ha nagysebességű internetkapcsolattal vagy birkatürelemmel rendelkezel, a java.sun.com címen is böngészheted őket, de hidd el nekünk, jobban jársz, ha a leírások a merevlemezeden vannak.

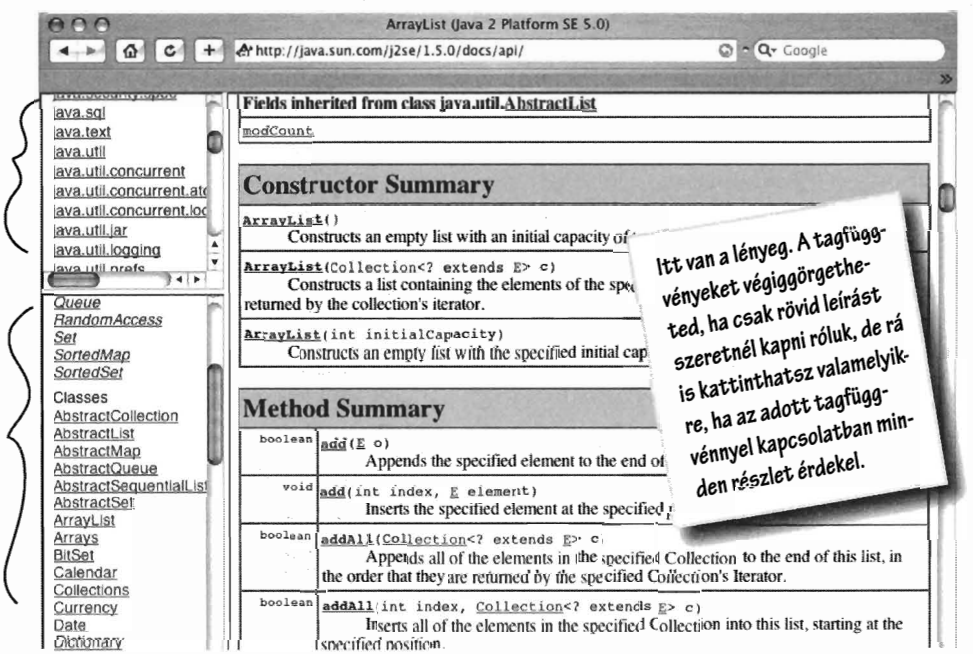
Az API-dokumentáció a legjobb forrás, amelyből többet tudhatsz meg egy osztályról és a tagfüggvényeiről. Tegyük fel, hogy egy kézikönyvet lapozgatva rábukkantál egy Calendar (Naptár) nevű osztályra a java.util csomagban. A könyv nem árul el róla túl sokat, csak annyit, hogy tudd, erre az osztályra van szükséged – a tagfüggvényeiről azonban több ismeretre lenne szükséged.

A kézikönyv például elmondja, hogy milyen argumentumokat várnak a tagfüggvények, és mit adnak vissza. Nézd meg mondjuk az ArrayList-et: a kézikönyvben megtalálod az indexOf() tagfüggvényt, amelyet a DotCom osztályban használtunk, de ha csak annyit tudnál, hogy van egy indexOf() tagfüggvény, amely egy objektumot vár, és az objektum sorszámát adja vissza (egy int érték formájában), még mindig szükséged lenne egy létfontosságú információra: mi történik, ha az objektum nincs benne a tömblistában? A tagfüggvény alakja („aláírása") önmagában nem árulja el, hogyan működik a tagfüggvény – az API-dokumentáció azonban igen (legalábbis legtöbbször). Az API-leírásból megtudhatod, hogy az indexOf() tagfüggvény –1-et ad vissza, ha a paraméterként átadott objektum nem szerepel az ArrayList-ben. Innen tudtuk mi is, hogy egyszerre használhatjuk arra, hogy megvizsgáljuk, hogy egy objektum egyáltalán *benne van-e* a tömblistában, illetve hogy egyidejűleg kiolvassuk a sorszámát, ha igen.

Az API-dokumentáció nélkül azonban azt hihettük volna, hogy az indexOf() tagfüggvény bekrepál, ha a tömblista nem tartalmazza az objektumot.

1 Görgess végig a csomagokon, és (rákattintva) jelölj ki egyet, hogy az alsó keretben csak az adott csomag osztályai jelenjenek meg.

2 Görgess végig az osztályokon, és (rákattintva) jelölj ki egyet. A fő böngészőablakban ekkor a kiválasztott osztály leírása jelenik meg.





Gyakorlat

Kódmágnesek



Össze tudod rakni úgy a kódrészleteket, hogy előálljon belőlük a lap alján látható kimenetet eredményező Java-program?

MEGJEGYZÉS: Ennek a gyakorlatnak az elvégzéséhez szükséged van egy ÚJ információra – ha megnézed az ArrayList leírását az API-dokumentációban, egy *másik* add tagfüggvényt is találsz, amely két argumentumot vár:

```
add(int index, Object o)
```

Ez a tagfüggvény lehetővé teszi, hogy megadd az ArrayList-nek, hogy *hová* tegye a neki átadott objektumot.

```
a.remove(2);
printAL(a);
printAL(a);
a.add(0,"zero");
a.add(1,"one");
public static void printAL(ArrayList<String> al) {
if (a.contains("two")) {
    a.add("2.2");
}
a.add(2,"two");
public static void main (String[] args) {
System.out.print(element + "  ");
}
System.out.println(" ");
if (a.contains("three")) {
    a.add("four");
}
public class ArrayListMagnet {
if (a.indexOf("four") != 4) {
    a.add(4, "4.2");
}
}
}
}
import java.util.*;
printAL(a);
ArrayList<String> a = new ArrayList<String>();
for (String element : al) {
a.add(3,"three");
printAL(a);
```

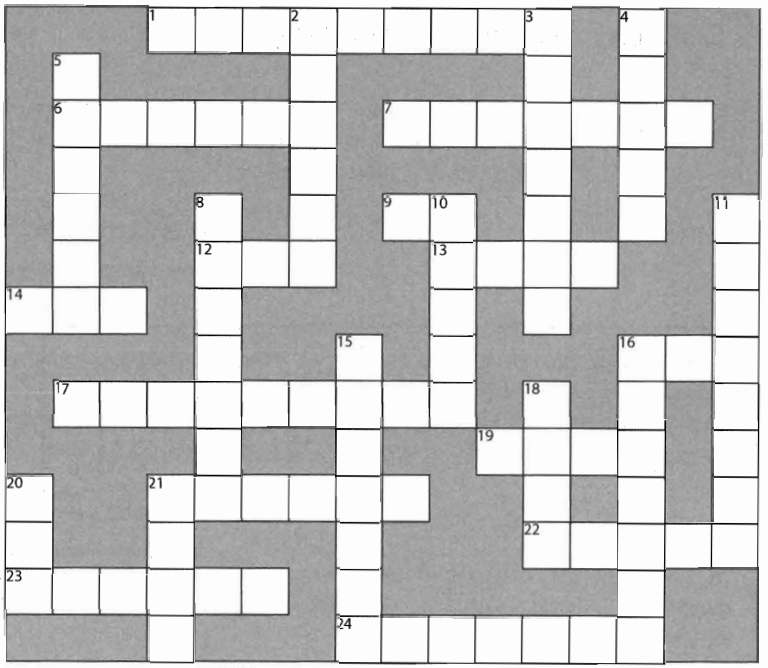
```
File  Edit  Window  Help  Dance
% java ArrayListMagnet
zero  one  two  three
zero  one  three  four
zero  one  three  four  4.2
zero  one  three  four  4.2
```



Java-keresztrejtvény 7.0

Hogyan segíthet egy keresztrejtvény a Java megtanulásában? Nos, a rejtvényben szereplő szavak **mindegyike** a Javával kapcsolatos (egy kakukktojástól eltekintve).

Tipp: Ha nem vagy biztos a dolgodban, jusson eszedbe az ArrayList!



Vízszintes

1. Nem tud viselkedni
6. Erről *tárgyalhatunk* (hehe, angolul)
7. Ott a pont
9. Egy elágazás kezdete
12. Tömblistát növel
13. Nagyon hosszú
14. Értéket másol
16. Nem objektum
17. Szteroidokkal teletömött tömb
19. Kiterjedés (angolul)
21. A 19. párja (angolul)
22. Spanyol harapnivaló (Megjegyzés: ennek semmi köze a Javához.)
23. Lusta ujjaknak
24. A csomagok helye (angolul)

Függőleges

2. A Java-akciók helye (angolul)
3. Címezhető egység (angolul)
4. A második legkisebb
5. A törtszámok alapértelmezett típusa
8. A könyvtár legnagyobb egysége
10. Alacsony sűrűségűnek kell lennie
11. Benne van valahol
15. Látszólagos (angolul)
16. Űrt kutató tagfüggvény
18. A bevásárlás és a tömbök közös pontja (angolul)
20. Könyvtár – betűszóval
21. Körbeforgó (angolul)

További tippek:

Vízszintes
1. 8 fajtája van
7. Gondolj az ArrayList-re!
16. Gyakran használatos alaptípus
21. Tömb kiterjedése
22. Nem Java-cucc – spanyol előétel

Függőleges
2. Mit írhatsz felül?
3. Gondolj az ArrayList-re!
4. és 10. Alaptípus
16. Gondolj az ArrayList-re!
18. Ezt kell készíteni hozzá



Gyakorlat – Megoldás

```
File  Edit  Window  Help  Dance
% java ArrayListMagnet
zero  one  two  three
zero  one  three  four
zero  one  three  four  4.2
zero  one  three  four  4.2
```

```
import java.util.*;

public class ArrayListMagnet {

  public static void main (String[] args) {

    ArrayList<String> a = new ArrayList<String>();

    a.add(0,"zero");
    a.add(1,"one");

     a.add(2,"two");

    a.add(3,"three");
    printAL(a);

    if (a.contains("three")) {
       a.add("four");
    }

    a.remove(2);

    printAL(a);

     if (a.indexOf("four") != 4) {
         a.add(4, "4.2");
     }

    printAL(a);

     if (a.contains("two")) {
        a.add("2.2");
     }

    printAL(a);
  }

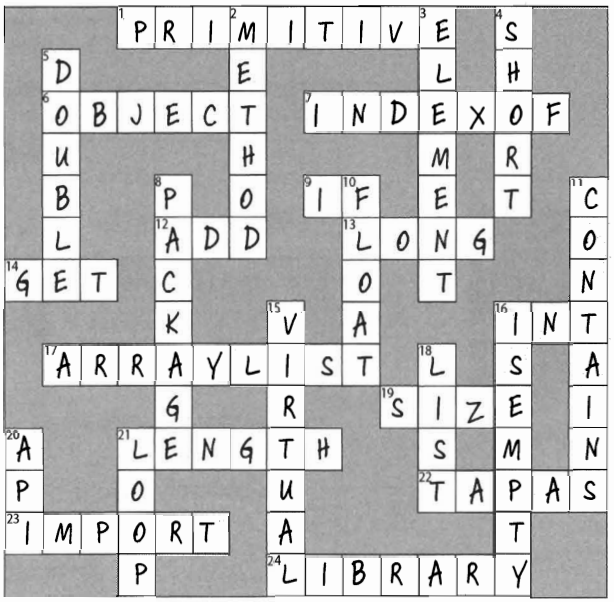
  public static void printAL(ArrayList<String> al) {

    for (String element : al) {

      System.out.print(element + "  ");
      }
    System.out.println(" ");
  }
}
```



Keresztrejtvény – Megoldás





Hegyezd ki a ceruzád!

Írj SAJÁT meghatározásokat! Nézd meg mindegyik szót, és próbálj saját meghatározást írni hozzá – könnyebbet vagy nehezebbet, vagy esetleg szakmailag pontosabbat, mint amit mi írtunk.

Vízszintes

1. _____
6. _____
7. _____
9. _____
12. _____
13. _____
14. _____
16. _____
17. _____
19. _____
21. _____
22. _____
23. _____
24. _____

Függőleges

2. _____
3. _____
4. _____
5. _____
8. _____
10. _____
11. _____
15. _____
16. _____
18. _____
20. _____
21. _____

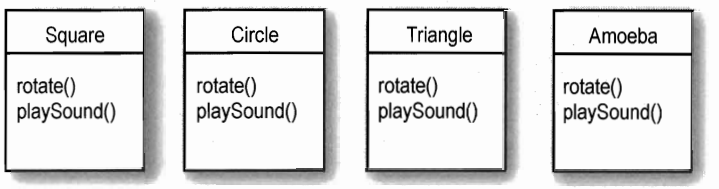
Objektumfalván szebb az élet



Gondolj a jövőre, amikor megtervezed a programjaidat! Ha lenne rá mód, hogy úgy írhasd meg a Java-kódjaidat, hogy több szabadidőd legyen, mennyit érne meg ez neked? Mi lenne, ha olyan kódot írhatnál, amit *mások* **könnyen** kibővíthetnek? És ha olyan kódot írhatnál, ami rugalmasan hozzáigazítható azokhoz a bosszantó változtatásokhoz, amelyeket az utolsó percben szoktak eszközölni a terven? Érdekelne a dolog? Ha igen, akkor szerencsés napod van. Mindössze 3 x 60 percet kell rászánnod, és mindez a tiéd lehet. Ha belépsz a Többalakúsági Programba, megtanulhatod a jobb osztálytervezés 5 lépését, a többalakúság (polimorfizmus, polymorphism) 3 fogását, a rugalmas kód készítésének 8 módját – és ha most rögtön feliratkozol, akkor egy bónusz leckét is kapsz 4 jótanáccsal az öröklés (inheritance) kiaknázásához! Ne késlekedj – ez a kihagyhatatlan ajánlat megadja neked azt a tervezői szabadságot és programozói rugalmasságot, amit megérdemelsz. Gyors, egyszerű, és máris megszerezhető. Lépj be ma, és megdobunk még egy szintnyi elvonatkoztatással!

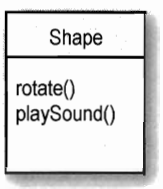
Székek háborúja – visszapörgetve

LARRY: Ismétlődik a kódod! A rotate eljárás mind a négy alakzatizében benne van! Ez hülye szerkezet: négy különböző rotate „tagfüggvényt" kell fenntartanod! Mi a jó ebben?
BRAD: Ó, ezek szerint nem láttad a végső tervet. Gyere Larry, megmutatom, hogy működik az objektumközpontú **öröklés**!

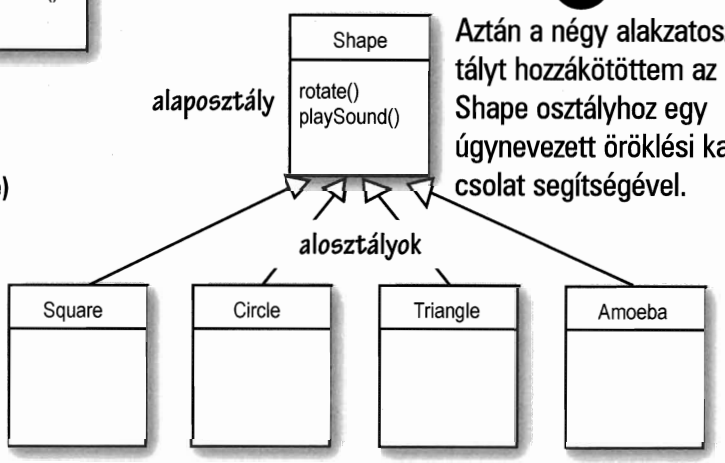


1 Megnéztem, mi a közös a négy osztályban.

2 Mindegyik alakzat (Shape), és mind forog (rotate), illetve hangot játszik le (playSound). Így hát elvonttá tettem ezeket a közös vonásokat, és egy új, Shape nevű osztályba tettem őket.



3 Aztán a négy alakzatosztályt hozzákötöttem az új Shape osztályhoz egy úgynevezett öröklési kapcsolat segítségével.



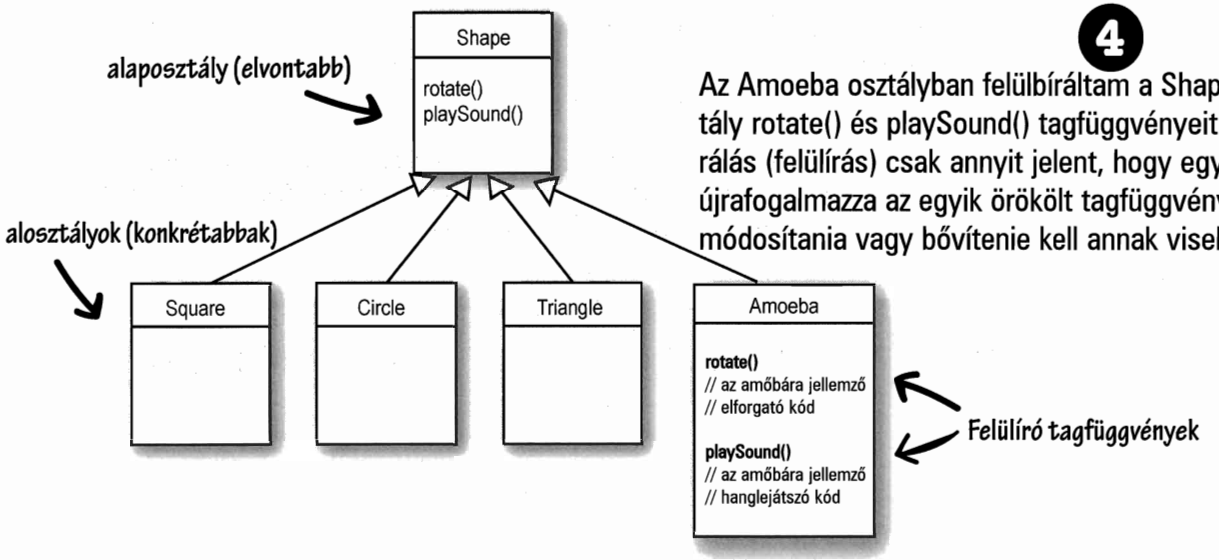
Ezeket úgy lehet olvasni, hogy **„a Négyzet (Square) örököl az Alakzattól (Shape)"**, **„a Kör (Circle) örököl az Alakzattól"**, és így tovább. A rotate() és playSound() tagfüggvényeket eltávolítottam a négy alakzatosztályból, így már csak egy példányt kell fenntartani.
A Shape osztály a négy másik osztály **alaposztálya** (ősosztálya, szuperosztálya; superclass), azok pedig **alosztályai** (subclass) a Shape-nek.
Az alosztályok öröklik az alaposztályuk tagfüggvényeit – más szavakkal, *ha a Shape osztály rendelkezik valamilyen vonással, akkor az az alosztályokban is automatikusan megtalálható lesz.*

Mi a helyzet az amőba forgatásával?

LARRY: De hát nem az volt a probléma, hogy az amőba alakzatnak teljesen más forgató és hanglejátszó eljárásra volt szüksége? Hogyan csinálhat mást az amőba, ha a viselkedését a Shape osztálytól *örökli*?

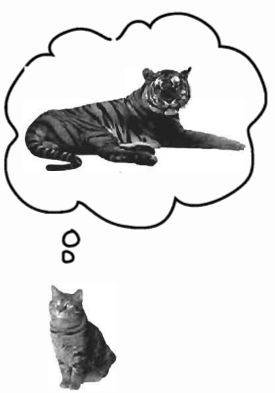
BRAD: Ez az utolsó lépés. Az Amoeba osztály **felülbírálja** (override) a Shape osztály tagfüggvényeit. Így futáskor a JVM pontosan tudja, hogy melyik *rotate()* tagfüggvényt kell elindítania, amikor azt az utasítást kapja, hogy forgassa el az amőbát.





4

Az Amoeba osztályban felülbíráltam a Shape alaposztály rotate() és playSound() tagfüggvényeit. A felülbírálás (felülírás) csak annyit jelent, hogy egy alosztály újrafogalmazza az egyik örökölt tagfüggvényét, ha módosítania vagy bővítenie kell annak viselkedését.



Agytorna

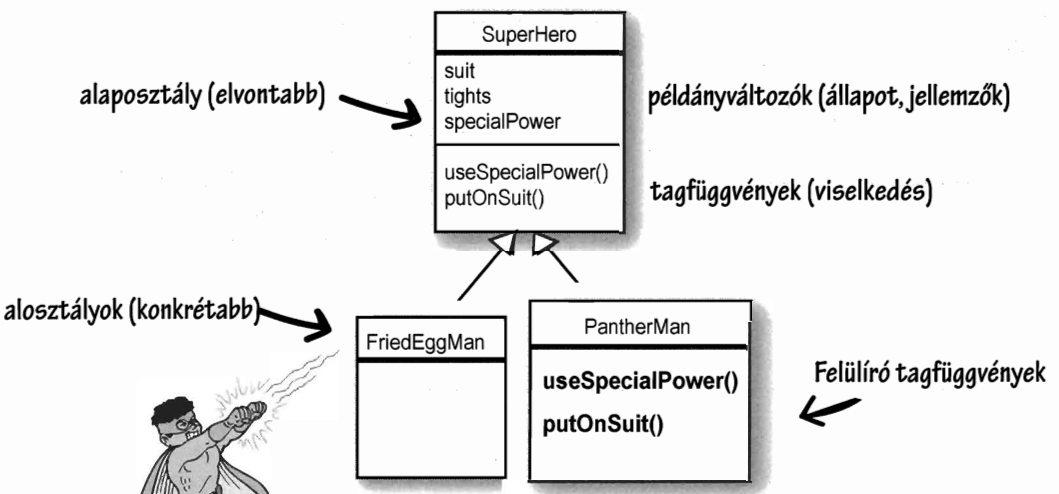
Hogyan ábrázolnál egy házimacskát és egy tigrist egy öröklési szerkezetben? A házimacska a tigris specializált változata? Melyikük lenne az alosztály, és melyik az alaposztály? Vagy mindkettő egy *másik* osztály alosztálya?

Hogyan terveznél meg egy öröklési szerkezetet? Milyen tagfüggvényeket bírálnál felül?

Gondold végig ezeket, *mielőtt* lapozol.

Az öröklés működése

Amikor öröklést alkalmazol, közös kódot teszel egy osztályba, és közlöd a konkrétabb osztályokkal, hogy a közös kódot tartalmazó (elvontabb) osztály az alaposztályuk. Amikor egy osztály egy másiktól örököl, **az alosztály örökli az alaposztály (ősosztály) képességeit**.
A Javában azt mondjuk, hogy az **alosztály *bővíti* (extends) az alaposztályt**.
Az öröklési kapcsolat azt jelenti, hogy az alosztály örökli az alaposztály **tagjait** (member). Amikor azt mondjuk, hogy „egy osztály tagjai", akkor a példányváltozókat és tagfüggvényeket értjük alatta.
Például ha a PantherMan (Párducember) a SuperHero (Szuperhős) alosztálya, akkor PantherMan automatikusan örökli az összes szuperhősre jellemző példányváltozót és tagfüggvényt; az olyanokat, mint a `suit` (jelmez), a `tights` (harisnya), a `specialPower` (kulonlegesKepesseg), a `useSpecialPower()` (hasznalKulonlegesKepesseg), és így tovább. A PantherMan **alosztály azonban új, saját tagfüggvényeket és példányváltozókat is tartalmazhat**, és **felülbírálhatja (felülírhatja) a** SuperHero **alaposztálytól örökölt tagfüggvényeket**.





FriedEggMan (Tükörtojásember) nem igényel semmilyen egyedi viselkedést, ezért nem ír felül egyetlen tagfüggvényt sem; a SuperHero tagfüggvényei és példányváltozói elegendőek a számára. PantherMan ezzel szemben egyedi igényeket támaszt a jelmezével és a különleges képességeivel kapcsolatban, ezért a PantherMan osztály mind a `useSpecialPower()`, mind a `putOnSuit()` (felveszJelmez) tagfüggvényt felülbírálja.
A példányváltozók nem íródnak felül, mert nincs rá szükség. Nem határoznak meg semmilyen különleges viselkedést, ezért egy alosztály olyan értéket adhat egy örökölt példányváltozónak, amilyet csak akar. PantherMan lilára festheti az örökölt harisnyáját, míg FriedEggMan nyugodtan hordhat fehéret.

Példa az öröklésre:

```
public class Doctor {

    boolean worksAtHospital;

    void treatPatient() {
      // vizsgálat
    }
}
```

```
public class FamilyDoctor extends Doctor {

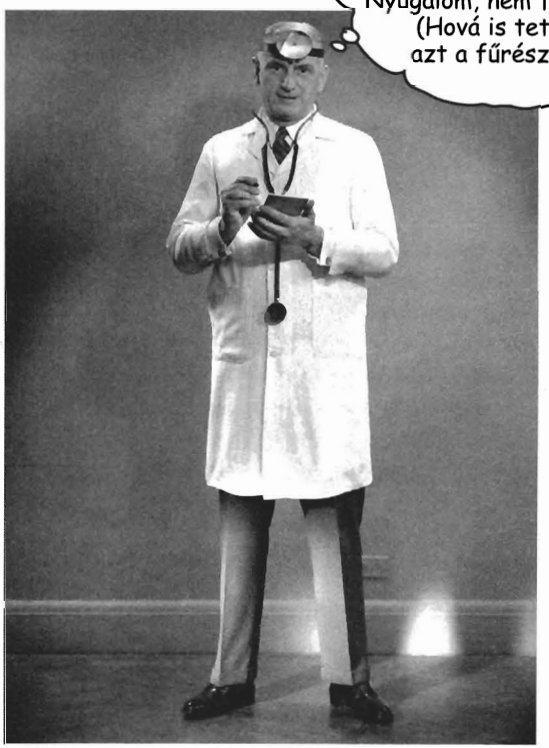
    boolean makesHouseCalls;
    void giveAdvice() {
      // tanácsadás
    }

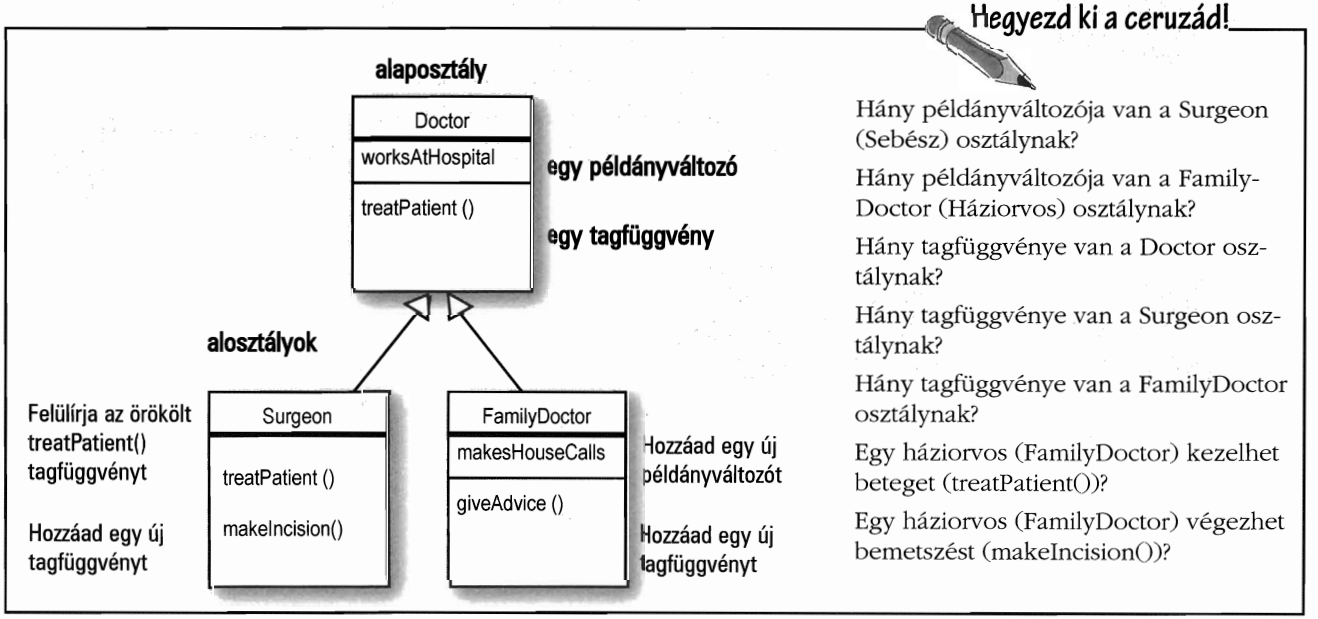
}
```

```
public class Surgeon extends Doctor{

   void treatPatient() {
      // műtét
   }

   void makeIncision() {
      // bemetszés (aúú!)
   }
}
```





Hegyezd ki a ceruzád!

Hány példányváltozója van a Surgeon (Sebész) osztálynak?

Hány példányváltozója van a FamilyDoctor (Háziorvos) osztálynak?

Hány tagfüggvénye van a Doctor osztálynak?

Hány tagfüggvénye van a Surgeon osztálynak?

Hány tagfüggvénye van a FamilyDoctor osztálynak?

Egy háziorvos (FamilyDoctor) kezelhet beteget (treatPatient())?

Egy háziorvos (FamilyDoctor) végezhet bemetszést (makeIncision())?

Tervezd meg egy állatszimulációs program öröklési fáját!

Tegyük fel, hogy azt a feladatot kapod, hogy tervezz egy szimulációs programot, amelyben a felhasználó különféle állatokat helyezhet el, hogy lássa, mi történik. A program kódját még nem kell megírnod, most a felépítés a lényeg.

Kapsz egy listát, amelyen szerepel *néhány* állat a program leendő kínálatából, de nem az összes. Azt tudod, hogy minden állatot egy-egy objektum fog jelképezni, és hogy az objektumok a környezetben mozogva olyan dolgokat fognak csinálni, amilyenekre az adott állatfajtát (objektumtípust) beprogramozták.

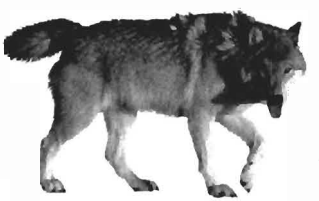
Ezenkívül lehetővé kell tenned a többi programozónak, hogy bármikor új állatfajtákat adjon a programhoz.

Először ki kell derítened, hogy mik azok a közös, elvont tulajdonságok, amelyekkel minden állat rendelkezik, és ezeket a tulajdonságokat be kell építened egy osztályba, amelyet minden állatosztály bővíthet.



1 Keresd meg a hasonló tulajdonságokkal és viselkedéssel rendelkező objektumokat!

Mi a közös ebben a hat állatfajtában? Ha rájössz, az segít a viselkedésük elvont-tá tételében (2. lépés). Milyen kapcsolatban állnak egymással ezek az állatfajták? Ez az öröklési fa felrajzolásához ad segítséget (4-5. lépés).











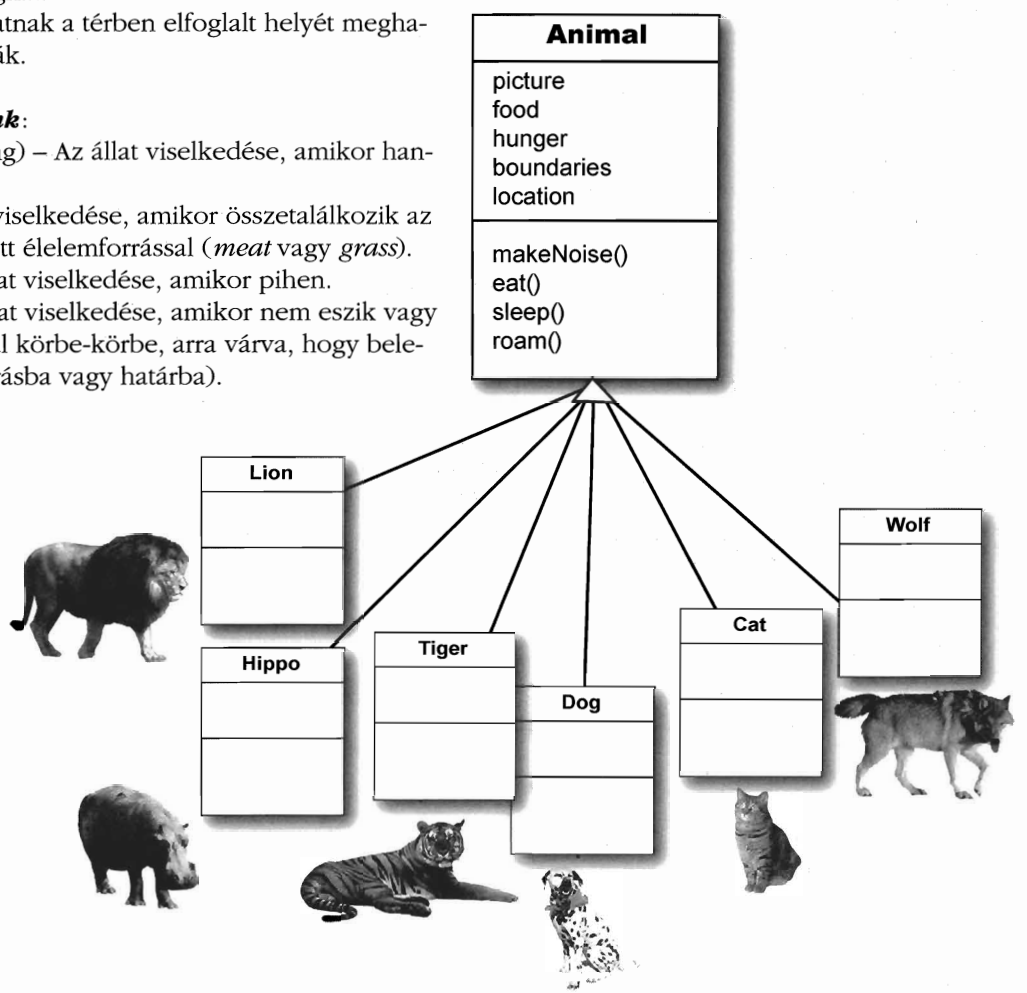
A kódismétlés elkerülése az alosztályokban öröklés segítségével

Öt ***példányváltozónk*** van:
picture (kép) – Az állat JPEG-képének fájlneve.
food (élelem) – Az az élelemtípus, amit az adott állat eszik. Pillanatnyilag csak két érték közül választhatsz: *meat* (hús) és *grass* (fű).
hunger (éhség) – Az állat éhségszintjét jelképező egész érték, ami attól függően változik, hogy az állat mikor (és mennyit) eszik.
boundaries (határok) – Annak a (mondjuk 640 x 480-as) „térnek" a szélességét és magasságát jelképező értékek, amelyben az állat mozoghat.
location (hely) – Az állatnak a térben elfoglalt helyét meghatározó X és Y koordináták.

És négy ***tagfüggvényünk***:
makeNoise() (kiadHang) – Az állat viselkedése, amikor hangot kell kiadnia.
eat() (eszik) – Az állat viselkedése, amikor összetalálkozik az általa előnyben részesített élelemforrással (*meat* vagy *grass*).
sleep() (alszik) – Az állat viselkedése, amikor pihen.
roam() (jarkal) – Az állat viselkedése, amikor nem eszik vagy alszik (hanem csak járkál körbe-körbe, arra várva, hogy beleütközzön egy élelemforrásba vagy határba).

2 Tervezz egy osztályt, amely a közös állapotot és viselkedést ábrázolja!

Minden objektumunk állat, ezért a közös alaposztály neve Animal (Állat) lesz. Ebbe az osztályba olyan tagfüggvényeket és példányváltozókat teszünk, amelyekre minden állatnak szüksége lehet.



Minden állat ugyanúgy eszik?

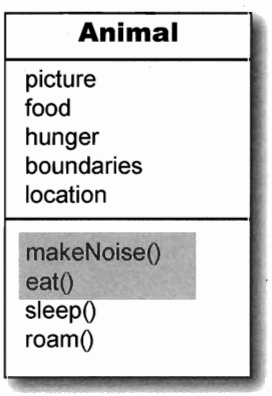
Tegyük fel, hogy egy dologban mindenki egyetért: a példányváltozók *minden* állattípus esetében működnek. Természetesen az oroszlán saját értékeket határoz meg a kép, az élelem (az oroszlán esetében ez a *hús*), az éhség, a határok és a hely számára, a víziló példányváltozói pedig megint más értékeket tartalmaznak, de a változói attól még ugyanazok, mint a többi állatfajta esetében. Ugyanez áll a kutyára, a tigrisre, és így tovább. De mi a helyzet a *viselkedéssel*?

Mely tagfüggvényeket kell felülbírálnunk?

Egy oroszlán ugyanolyan **hangot** ad ki, mint egy kutya? Egy macska ugyanúgy **eszik**, mint egy víziló? A *te* programodban talán igen, de a miénkben az evés és a kiadott hang állatfüggő. Arról viszont fogalmunk sincs, hogyan lehetne ezeket a tagfüggvényeket úgy kódolni, hogy bármely állat esetében működjenek. Na jó, ez nem igaz. Megírhatnánk például úgy a makeNoise() tagfüggvényt, hogy mindössze annyit csináljon, hogy lejátszik egy hangfájlt, amelyet egy példányváltozó határoz meg az adott állatfajta számára, de ez elég egysíkú viselkedést eredményezne. Egyes állatok különböző helyzetekben (például amikor esznek vagy amikor ellenségbe botlanak) más-más hangot adhatnak ki. Ugyanazt kell tenned az Animal alosztályaiban, mint amit az Amoeba osztályban tettünk, amikor felülbíráltattuk vele a Shape osztály rotate() tagfüggvényét, hogy az amőbára jellemző (más szóval, *egyedi*) viselkedést határozzon meg.







Az eat() és a makeNoise() tagfüggvényeket célszerű felülbírálni, hogy minden állatfajta sajátos viselkedést határozhasson meg az evéshez és a hangkibocsátáshoz. Egyelőre úgy tűnik, a sleep() és a roam() általános maradhat.

3 **Döntsd el, hogy az alosztályoknak szükségük van-e csak rájuk jellemző viselkedésekre (tagfüggvény-megvalósításokra)!**

Az Animal osztályt megvizsgálva úgy találtuk, hogy az eat() és a makeNoise() tagfüggvényeket felül kell bírálniuk az egyes alosztályoknak.

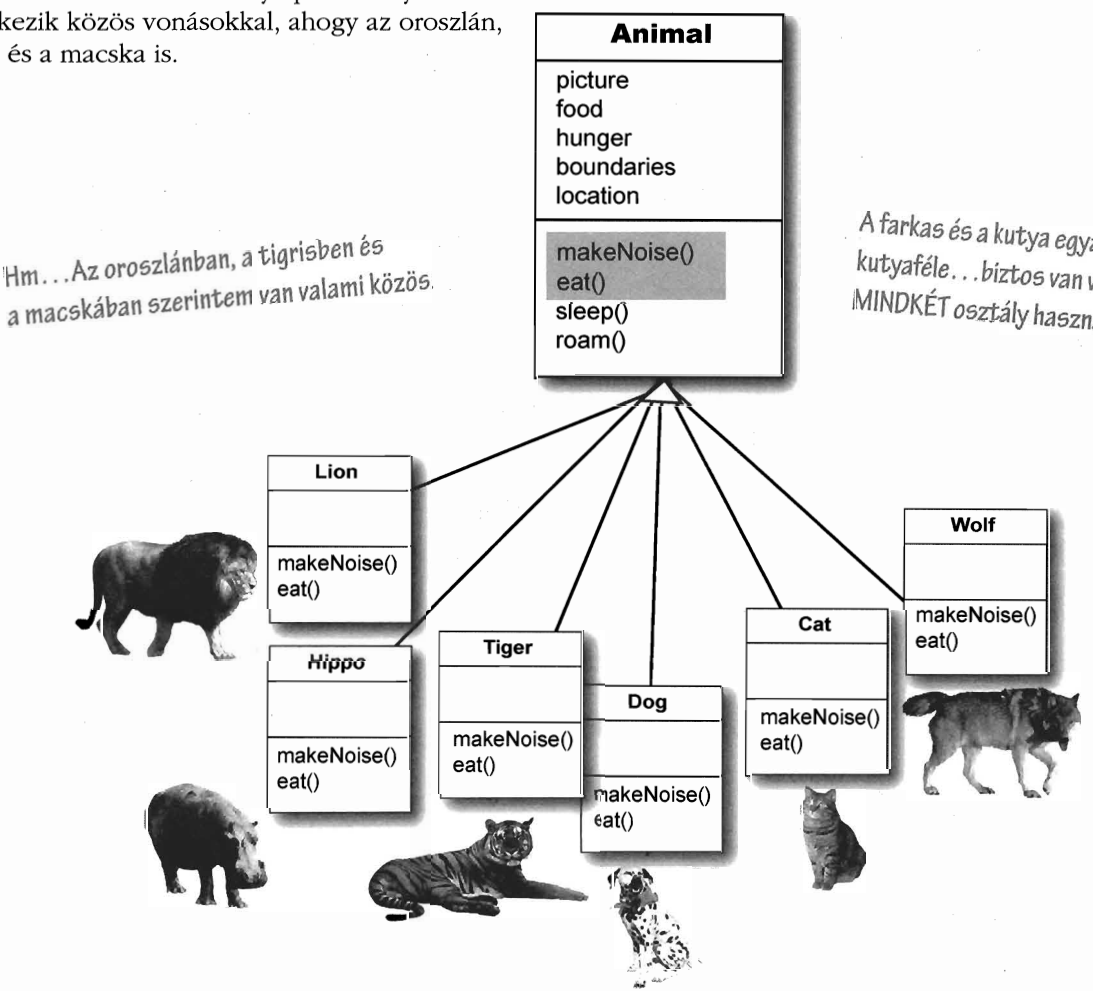
További öröklési lehetőségek keresése

Az osztályhierarchia kezd alakot ölteni. Minden alosztály felülírja a *makeNoise()* és az *eat()* tagfüggvényt, hogy a kutyaugatást ne lehessen összekeverni a macskanyávogással (ami mindkét fél számára sértő lenne), és a víziló ne ugyanazt egye, mint az oroszlán.

Ennél azonban talán többet is tehetsz. Ha megnézed az Animal alosztályait, észreveheted, hogy egyeseket össze lehet csoportosítani, és lehetnek olyan kódrészek, amelyek csak *erre* az új csoportra érvényesek. A farkas és a kutya például nyilván rendelkezik közös vonásokkal, ahogy az oroszlán, a tigris és a macska is.

4 Keress további lehetőségeket az elvont ábrázolásra két vagy több olyan alosztály meghatározásával, amelyeknek közös viselkedésre lehet szüksége!

Az osztályainkra pillantva felfedezheted, hogy a Wolf (Farkas) és a Dog (Kutya) osztozhat egyes viselkedéseken, és ugyanez áll a Lion (Oroszlán), Tiger (Tigris) és Cat (Macska) osztályokra is.



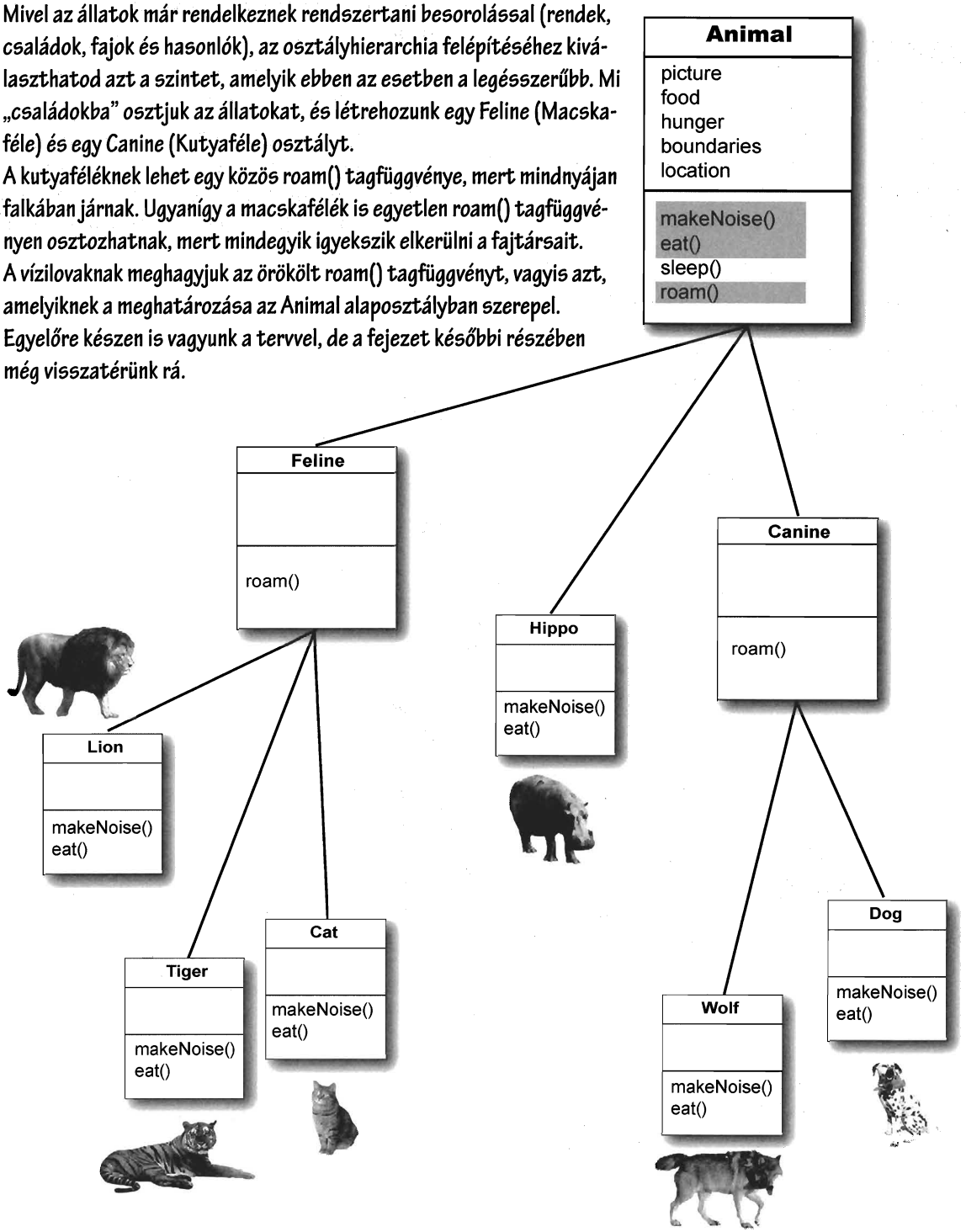
5 Fejezd be az osztályhierarchia felépítését!

Mivel az állatok már rendelkeznek rendszertani besorolással (rendek, családok, fajok és hasonlók), az osztályhierarchia felépítéséhez kiválaszthatod azt a szintet, amelyik ebben az esetben a legésszerűbb. Mi „családokba" osztjuk az állatokat, és létrehozunk egy Feline (Macskaféle) és egy Canine (Kutyaféle) osztályt.

A kutyaféléknek lehet egy közös roam() tagfüggvénye, mert mindnyájan falkában járnak. Ugyanígy a macskafélék is egyetlen roam() tagfüggvényen osztozhatnak, mert mindegyik igyekszik elkerülni a fajtársait.

A vízilovaknak meghagyjuk az örökölt roam() tagfüggvényt, vagyis azt, amelyiknek a meghatározása az Animal alaposztályban szerepel.

Egyelőre készen is vagyunk a tervvel, de a fejezet későbbi részében még visszatérünk rá.



Melyik tagfüggvény hívódik meg?

A Wolf osztálynak négy tagfüggvénye van. Egyet az Animal osztálytól örököl, egyet a Canine-tól (ami valójában az Animal osztály egyik tagfüggvényének felülbírált változata), két további tagfüggvényt pedig felülír. Amikor létrehozol egy Wolf objektumot, és értékül adod egy változónak, ez után a hivatkozó változó után a pont műveleti jelet kitéve mind a négy tagfüggvényt meghívhatod. De melyik *változata* is hívódik meg az említett tagfüggvényeknek?

új Wolf objektum létrehozása

```
Wolf w = new Wolf();
```

a Wolf-féle változat meghívása

```
w.makeNoise();
```

a Canine-féle változat meghívása

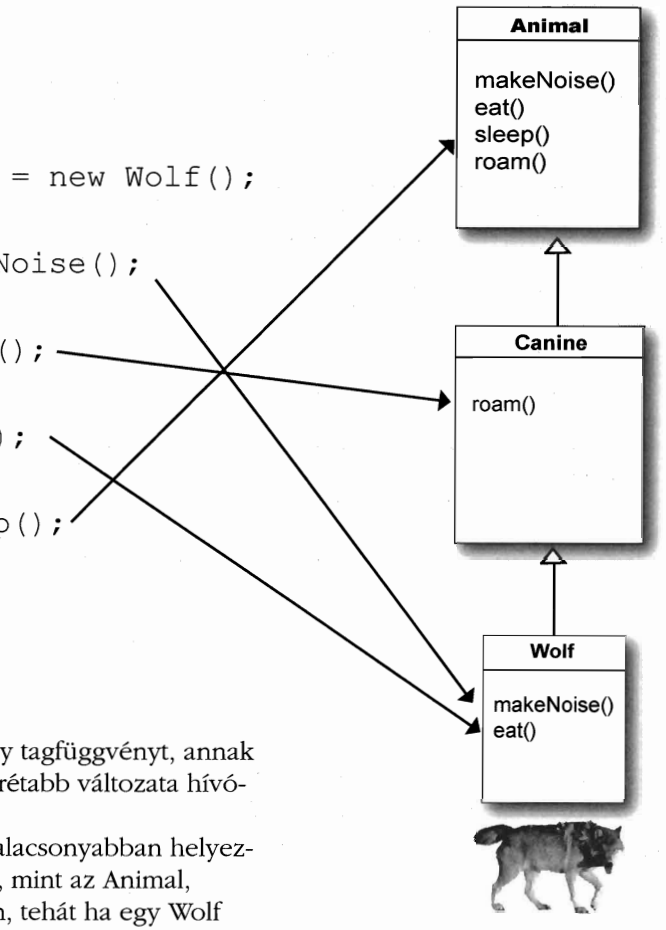
```
w.roam();
```

a Wolf-féle változat meghívása

```
w.eat();
```

az Animal-féle változat meghívása

```
w.sleep();
```



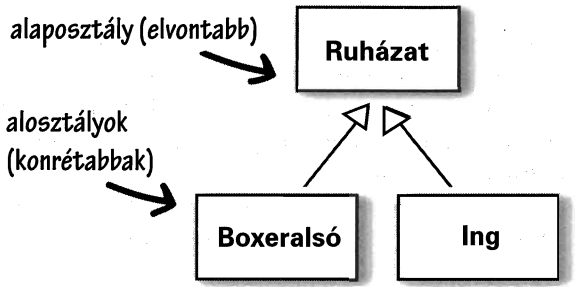
Amikor egy objektumhivatkozásra hívsz meg egy tagfüggvényt, annak mindig az adott objektumtípusra nézve legkonkrétabb változata hívódik meg. Más szavakkal, a ***„legalsó" győz!***

A „legalsó" itt azt jelenti, hogy az, amelyik a legalacsonyabban helyezkedik el az öröklési fában. A Canine lejjebb van, mint az Animal, a Wolf pedig még a Canine-nál is alacsonyabban, tehát ha egy Wolf objektumra mutató hivatkozásra hívsz meg egy tagfüggvényt, a JVM először a Wolf osztályban fogja keresni azt. Ha a JVM nem talál változatot a tagfüggvényből a Wolf osztályban, addig lépked felfelé az öröklési hierarchiában, amíg rá nem bukkan egyre.

Öröklési fa tervezése

Osztály	Alaposztályok	Alosztályok
Ruházat	---	**Boxeralsó, Ing**
Boxeralsó	**Ruházat**	
Ing	**Ruházat**	

Öröklési táblázat



Öröklési osztálydiagram



Hegyezd ki a ceruzád!

Keress értelmes kapcsolatokat, és töltsd ki az utolsó két oszlopot!

Ide rajzold az öröklési diagramot.

Osztály	Alaposztályok	Alosztályok
Zenész		
Rocksztár		
Rajongó		
Basszusgitáros		
Koncertzongorista		

Tipp: nem mindent lehet valami máshoz kapcsolni.
Tipp: a listát kiegészítheted és a felsorolt osztályokat módosíthatod is.

Nincsenek

hülye kérdések

K: Azt mondtátok, hogy attól az osztálytípustól kezdve, amelyikre meghívtam a tagfüggvényt (az előző oldal példájában ez a Wolf osztály volt) a JVM felfelé lépked az öröklési fában. De mi történik, ha a JVM sehol nem találja a tagfüggvényt?

V: Jó kérdés, de emiatt nem kell aggódnod. A fordító garantálja, hogy egy adott tagfüggvény meghívható legyen egy bizonyos hivatkozástípusra, de azt nem mondja meg (és nem is foglalkozik vele), hogy futásidőben ténylegesen *melyik* osztályból származik majd a tagfüggvény. A farkasos példában a fordító egy sleep() tagfüggvényt keres, de az nem érdekli, hogy a sleep() meghatározása valójában az Animal osztályban található (tehát onnan öröklődik). Ne feledd, hogy ha egy osztály *örököl* egy tagfüggvényt, akkor *rendelkezik* vele. Az, hogy az örökölt tagfüggvényt hol (melyik alaposztályban) határozták meg, nem érdekli a fordítót. Futáskor azonban a **JVM mindig a megfelelő változatot választja ki**. A megfelelő változat pedig azt jelenti, hogy ***az adott objektumra nézve a legkonkrétabb***.

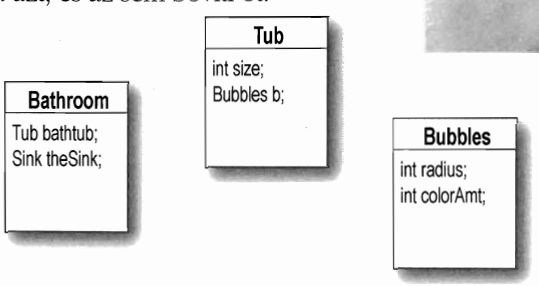
IS-A és HAS-A

Emlékezz vissza, hogy azt mondtuk, hogy amikor egy osztály örököl egy másiktól, akkor az alosztály *bővíti* (extends) az alaposztályt. Ha azt szeretnéd tudni, hogy egy dolog bővítése-e egy másiknak, végezd el az úgynevezett IS-A („egyfajta") vizsgálatot:

„Triangle IS-A Shape" (A Háromszög egyfajta Alakzat) – aha, ez működik.
„Cat IS-A Feline" (A Macska egyfajta Macskaféle) – ez is igaz.
„Surgeon IS-A Doctor" (A Sebész egyfajta Orvos) – még mindig rendben vagyunk.
„A Fürdőkád bővíti a Fürdőszobát" – ez is ésszerűnek hangzik.
Amíg el nem végzed az IS-A tesztet.

Ha tudni szeretnéd, hogy helyesen tervezted-e meg a típusaidat, tedd fel a kérdést: „Mondhatom, hogy az X típus egyfajta Y?". Ha nem, akkor tudhatod, hogy valami gond van a felépítéssel: „Tub IS-A Bathroom" (A Fürdőkád egyfajta Fürdőszoba) – hát ez egyértelműen hibás. És ha megfordítjuk a dolgot, hogy a Bathroom osztály bővítse a Tub-ot? Ez sem jó: „Bathroom IS-A Tub" (A Fürdőszoba egyfajta Fürdőkád) – nem stimmel. A fürdőkád és a fürdőszoba között természetesen *van* valamilyen kapcsolat, de ez nem fejezhető ki örökléssel. A kettő között úgynevezett HAS-A (tartalmaz, van neki, birtokol, rendelkezik vele stb.) kapcsolat áll fenn. Van értelme annak, hogy „Bathroom HAS-A Tub" (A Fürdőszoba tartalmaz Fürdőkádat)? Ha igen (márpedig igen), akkor az azt jelenti, hogy a Bathroom osztálynak van egy Tub példányváltozója. Más szavakkal, a Bathroom *hivatkozik* a Tub-ra, de nem *bővíti* azt, és az sem bővíti őt.





Bathroom HAS-A Tub (A Fürdőszoba tartalmaz Fürdőkádat) és Tub HAS-A Bubbles (A Fürdőkád tartalmaz Habot), de senki nem örököl senkitől (senki nem bővít senkit).

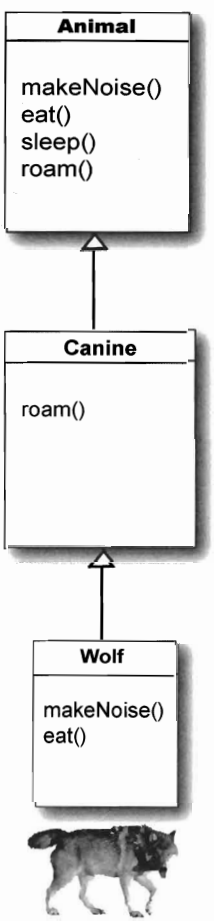
Várj csak! Tovább is van!

Az IS-A vizsgálat bárhol működik az öröklési fában. Ha az örklési fát jól tervezted meg, az IS-A tesztnek sikerrel kell járnia, *bármelyik* alosztályról kérdezed is meg, hogy olyan-e, mint *bármelyik* ősosztálya.

Ha B osztály bővíti A osztályt, akkor B egyfajta (IS-A) A. Ez az öröklési fában mindenütt igaz. Ha C osztály bővíti B osztályt, akkor C B-re és A-ra egyaránt teljesíti az IS-A tesztet.

Canine extends Animal
Wolf extends Canine
Wolf extends Animal

Canine IS-A Animal
Wolf IS-A Canine
Wolf IS-A Animal



Egy az itt láthatóhoz hasonló öröklési fa esetében mindig mondhatod azt, hogy **„Wolf extends Animal"** vagy hogy **„Wolf IS-A Animal"**. Az nem számít, hogy az Animal a Wolf alaposztályának alaposztálya. **Amíg az Animal az öröklési fában *valahol* a Wolf felett helyezkedik el, a „Wolf IS-A animal" állítás mindig igaz lesz.**

Az állatok öröklési fájának szerkezete ezt mondja: „A Farkas egyfajta Kutyafélé, tehát a Farkas bármit megtehet, amit egy Kutyafajta. Ezenkívül a Farkas egyfajta Állat, ezért a Farkas bármilyen dolgot csinálhat, amit egy Állat."

Nem számít, hogy a Wolf osztály felülbírálja-e az Animal vagy a Canine egyes tagfüggvényeit. Ami a világot (a máshol található kódok világát) illeti, a Wolf osztály mind a négy tagfüggvényt felhasználhatja. Az, hogy *miként*, vagy hogy *melyik osztályban írod felül őket*, nem érdekes. A Farkas kiadhat hangot (makeNoise), ehet (eat), aludhat (sleep) és járkálhat (roam), mert a Wolf az Animal osztály bővítése.



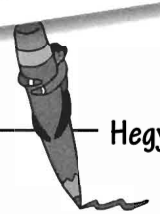
Honnan tudhatod, hogy az öröklés helyes-e?

Eddig nyilvánvalóan nem tértünk ki mindenre, de az objektumközpontú tervezés rejtelmeiről a következő fejezetben sokkal többet tudsz majd meg. (És az ebben a fejezetben elkészített terven is finomítani fogunk.)

Egyelőre azonban az IS-A vizsgálat elég, hogy utat mutasson. Ha az „X IS-A Y" állítás igaz, akkor a két osztály (X és Y) valószínűleg ugyanannak az öröklési hierarchiának a része, így esély van rá, hogy a viselkedésük ugyanaz, de legalábbis átfedés mutatkozik köztük.

Ne feledd, hogy az öröklési IS-A kapcsolat csak egy irányban működik!

A „Triangle IS-A Shape" (A Háromszög egyfajta Alakzat) állítás igaz, tehát a Triangle bővítheti a Shape osztályt. A fordítottja („Shape IS-A Triangle", vagyis Az Alakzat egyfajta Háromszög) azonban *nem* igaz, vagyis a Shape nem lehet a Triangle osztály bővítése. Jegyezd meg, hogy az IS-A kapcsolat maga után vonja, hogy ha „X IS-A Y", akkor X bármit megtehet, amit Y (sőt valószínűleg többet).



Hegyezd ki a ceruzád!

Pipáld ki azokat az állításokat, amelyek igazak!

- ❑ **A Sütő bővíti a Konyhát**
- ❑ **A Gitár bővíti a Hangszert**
- ❑ **A Személy bővíti az Alkalmazottat**
- ❑ **A Ferrari bővíti a Motort**
- ❑ **A Tükörtojás bővíti az Ételt**
- ❑ **A Beagle bővíti a Házikedvencet**
- ❑ **A Tároló bővíti a Csuport**
- ❑ **A Fém bővíti a Titániumot**
- ❑ **A GratefulDead bővíti a Zenekart**
- ❑ **A Szőke bővíti az Okost**
- ❑ **Az Ital bővíti a Martinit**

Tipp: alkalmazd az IS-A tesztet!

Nincsenek

hülye kérdések

K: Azt már láttuk, hogyan öröklik az alosztályok az alaposztályok tagfüggvényeit, de mi a helyzet, ha egy alosztály szeretné az egyik alosztályában levő tagfüggvény-változatot használni?

V: Az alaposztályok nem feltétlenül *tudnak* az alosztályaikról. Írhatsz például egy osztályt, amit aztán valaki jóval később bővít (alosztályokat származtat belőle). De még ha az alaposztály megalkotója tudomással bír is egy tagfüggvény alosztálybeli változatáról (és használni akarja azt), akkor sincs lehetőség *fordított* öröklésre. Gondolj csak bele: a gyerekek örökölnek tulajdonságokat a szülőktől, nem fordítva.

K: Mi a helyzet, ha egy alosztályban egy tagfüggvénynek az alaposztálybeli és az alosztálybeli felülíró változatát IS használni szeretném? Más szavakkal, nem akarom teljesen *lecserélni* az ősosztálybeli változatot, csak hozzá szeretnék adni ezt-azt.

V: Megteheted, sőt ez nagyon fontos lehetőség! Gondolj csak bele: az „extends" (bővít) szó azt jelenti, hogy „*bővíteni* akarod az alaposztály képességeit".

```
public void roam() {
      super.roam();
        // egyéni járkálás
  }
```

Az alaposztály tagfüggvényeit megtervezheted úgy, hogy az osztály olyan tagfüggvény-megvalósításokat tartalmazzon, amelyek bármely alosztályban működnek, még ha az alosztályoknak „hozzájuk is kell fűzniük" némi kódot. Az alosztálybeli felülíró tagfüggvényekből a **super** kulcsszóval hívhatod meg az ősosztálybeli változatot.
Ez olyan, mintha azt mondanád, hogy „először futtasd az alaposztálybeli változatot, majd térj vissza, és fejezd be az én kódommal..."

itt a roam() örökölt változatát hívjuk meg, majd visszatérünk az alosztály saját kódjának végrehajtásához

Ki kapja a Porschét, és ki a porcelánt?

(Honnan tudhatod, hogy egy alosztály mit örökölhet az ősétől?)



Az alosztályok öröklik az ősosztályuk tagjait. A tagok közé többek között a példányváltozók és a tagfüggvények tartoznak, de a könyv későbbi részében más örökölt tagokat is látni fogsz. Az alaposztályok dönthetik el, hogy egy adott alosztályuk örökölhet-e egy adott tagot, a taghoz biztosított hozzáférési szint megválasztásával.

Ebben a könyvben négy hozzáférési szinttel foglalkozunk.
A leginkább korlátozótól a leginkább megengedő felé haladva:

private **default** **protected** **public**

Megjegyzés: a default (alapértelmezett) és a protected (védett) szintről a 16. fejezetben és a B függelékben lesz szó bővebben.

A hozzáférési szintek szabályozzák, hogy *ki mit lát*, és létfontosságúak ahhoz, hogy jól tervezett, megbízható Java-kódot írj. Egyelőre összpontosítsunk a public (nyilvános) és a private (privát) szintre.
Az ezekre a szintekre vonatkozó szabályok egyszerűek:
A **nyilvános** tagok öröklődnek.
A **privát** tagok nem öröklődnek.
Amikor egy alosztály örököl egy tagot, az olyan, ***mintha maga az alosztály határozná meg azt***. A Shape osztály példájában a Square örökli a `rotate()` és `playSound()` tagfüggvényeket, de a külvilágnak (a máshol található kódoknak) ez egyszerűen annyit tesz, hogy a Square osztály tartalmaz egy `rotate()` és egy `playSound()` tagfüggvényt.
Egy osztály tagjai az osztályban meghatározott változók és tagfüggvények, valamint minden, amit az osztály egy alaposztályától örököl.

Mit szabad és mit nem szabad, amikor örökölsz?

Bár az alább felsorolt szabályok mögött megbúvó okok egy részét csak a könyv későbbi részében fedjük fel, néhány szabály puszta *ismerete* is segít, hogy helyesebb öröklési terveket készíts.

TEDD: Alkalmazz öröklést, ha az egyik osztály egy alaposztály konkrétabb típusa. Például: A Fűz *egyfajta* (is-a) Fa, ezért a Fűz osztály *bővítheti* (extends) a Fát.

TEDD: Érdemes lehet öröklést alkalmazni, ha bizonyos viselkedéseket (megvalósított kódokat) több, azonos általános típusba tartozó osztály közösen használhat. Például: a Négyzet, Kör és Háromszög alakzatok egyaránt forognak és hangot játszanak le, ezért ezeket a műveleteket célszerű egy Alakzat nevű alaposztályba tenni, ami a karbantartást és a bővítést is megkönnyíti. Ne feledd azonban, hogy bár az öröklés az objektumközpontú programozás egyik sarokköve, nem feltétlenül a legjobb módja a viselkedések újrahasznosításának. Az első lépések megtételében segít, és gyakran valóban helyes választás, de a tervezési minták sokszor kifinomultabb és rugalmasabb lehetőségeket nyújtanak. Ha nem ismered a programtervezési mintákat, érdemes lehet ez után a kötet után elolvasnod a *Head First Design Patterns* című könyvünket is.

NE TEDD: Ne alkalmazz öröklést csak azért, hogy újrahasznosíthasd egy másik osztály kódját, ha az alaposztály és az alosztály kapcsolata megsérti a fenti két szabály egyikét. Tegyük fel például, hogy írtál egy különleges nyomtatókódot az Alarm (Csengő) osztályban, és mivel a Piano (Zongora) osztályban is szükséged van nyomtatókódra, a Piano-val bővíted az Alarm-ot, hogy a Piano örökölje azt. Ennek azonban semmi értelme: a Piano *nem* az Alarm konkrétabb típusa. (A nyomtatókódot tehát egy Printer – Nyomtató – nevű osztályban kell elhelyezned, hogy a nyomtatható objektumok a „tartalmaz" (has-a) kapcsolaton keresztül vehessék igénybe a nyomtatókódot.)

NE TEDD: Ne alkalmazz öröklést, ha az alaposztály és az alosztály nem teljesíti az IS-A („egyfajta") tesztet. Mindig tedd fel magadnak a kérdést, hogy az alosztály konkrétabb típusa-e az alaposztálynak. Például: „A Tea EGYFAJTA Ital" – ez igaz; „Az Ital EGYFAJTA Tea" – ez viszont nem.



Újratöltve

- Az alosztályok *bővítik* az alaposztályukat.
- Az alosztályok az alaposztályuk minden *nyilvános* példányváltozóját és tagfüggvényét öröklik, de a *privát* példányváltozóit és tagfüggvényeit nem.
- Az örökölt tagfüggvények *felülbírálhatók*, a példányváltozók azonban *nem* (bár az alosztályban kaphatnak *új meghatározást*, de ez nem ugyanaz, és szinte soha nincs rá szükség).
- Ellenőrizd az öröklési hierarchia érvényességét az IS-A („egyfajta") teszttel. Ha X *bővíti* Y-t, akkor az „X *egyfajta* Y" állításnak igaznak kell lennie.
- Az IS-A kapcsolat csak egy irányban működik. A Víziló egyfajta Állat, de nem minden Állat Víziló.
- Amikor egy tagfüggvényt felülírsz egy alosztályban, majd a tagfüggvényt az alosztály egy példányára alkalmazod, a függvény felülírt változata hívódik meg. (*A legalsó győz.*)
- Ha B osztály bővíti A-t, és C bővíti B-t, akkor B osztály *egyfajta* A, C osztály *egyfajta* B, és C *egyfajta* A is.

Akkor milyen előnyei is vannak az öröklésnek?

Ha öröklést alkalmazol, az objektumközpontú tervezés számos előnyét kihasználhatod. Megszabadulhatsz a kódismétléstől, ha azokat a viselkedéseket, amelyek több osztályra is jellemzőek, elvonttá teszed, és egy alaposztályban helyezed el. Így ha módosítanod kell rajtuk, csak egy helyen kell változtatnod, és *a változás varázslatos módon az összes osztályban tükröződni fog, amely örökli az adott viselkedést.* Na jó, nincs benne semmilyen varázslat, viszont *nagyon* egyszerű: végrehajtod a módosítást, és újra lefordítod az osztályt. Ennyi az egész. Az **alosztályokhoz nem kell hozzányúlnod. Csak bedobod az alaposztály új változatát, és az azt bővítő valamennyi osztály automatikusan az új változatot fogja használni.** A Java-programok csupán egy halom osztályból állnak, ezért az alosztályokat nem kell újrafordítani ahhoz, hogy használhassák az alaposztály új változatát. Amíg az alaposztály *nem tesz tönkre* semmit az alosztályokban, minden rendben lesz. (A „tönkretesz" jelentését az adott környezetben a könyv későbbi részében tárgyaljuk. Egyelőre értsd úgy, hogy valami olyasmit módosítasz az alaposztályban, amire az alosztályok támaszkodnak, például egy bizonyos tagfüggvény argumentumait vagy visszatérési típusát, a tagfüggvény nevét, és így tovább.)

1 Elkerülöd a kódismétlést.

Tedd a közös kódokat egy helyre, és hagyd, hogy az alosztályok örököljék azokat egy alaposztálytól. Így ha az adott viselkedésen változtatni szeretnél, csak egy helyen kell módosítanod, és a változást mindenki (mármint az összes alosztály) látni fogja.

2 Közös protokollt határozol meg osztályok egy csoportja számára.



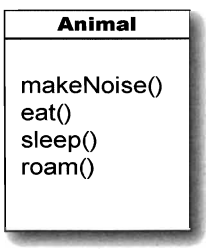
Az öröklés révén biztosíthatod, hogy az adott őstípus alá csoportosított összes osztály rendelkezzen az őstípus valamennyi tagfüggvényével.*

Más szavakkal, közös protokollt (szabályokat) határozol meg osztályok egy halmazára, mely osztályok öröklés révén kapcsolatban állnak egymással.

Amikor olyan tagfüggvényeket határozol meg egy alaposztályban, amelyeket az alosztályai örökölhetnek, egyfajta protokollt jelentesz be a többi kódnak, ami azt mondja, hogy „Minden altípusom (vagyis alosztályom) megteheti ezeket a dolgokat, ezekkel a tagfüggvényekkel, amelyek így néznek ki…"

Más szavakkal, egyfajta *szerződést* kötsz.

Az Animal osztály a következő protokollt vezeti be az összes altípusára:



Bejelented a világnak, hogy ezt a négy dolgot – a tagfüggvény-argumentumokat és a visszatérési típusokat is beleértve – bármelyik Állat megteheti.

Ne feledd, hogy amikor azt mondjuk, hogy *bármelyik Állat*, akkor ebbe az Animal osztályt és *az Animal-t bővítő valamennyi osztályt* beleértjük. Ez pedig annyit tesz, hogy *minden olyan osztályt, amely az öröklési hierarchiában valahol az Animal alatt áll.*

Az igazán klassz részhez azonban még nem érkeztünk el. A legjobbat – a *többalakúságot* (polimorfizmus, polymorphism) – a végére tartogattuk.

Ha őstípust határozol meg osztályok egy csoportja számára, *az adott őstípus bármelyik alosztályát behelyettesítheted ott, ahol a kód az őstípust várja.*

Tessék?

Ne aggódj, még bele sem fogtunk a magyarázatba. Csak két oldal, és szakértő leszel a témában.

*Amikor azt mondjuk, hogy „valamennyi tagfüggvény", akkor „az összes *örökölhető* tagfüggvényt" értjük alatta. Ez egyelőre annyit tesz, hogy „minden *nyilvános* tagfüggvény", bár később finomítunk egy kicsit ezen a meghatározáson.

És ez tetszik, mert…

Mert kihasználhatod a többalakúság előnyeit.

Ami azért fontos, mert…

Mert egy alosztályból létrehozott objektumra az őstípusként bevezetett hivatkozással hivatkozhatsz.

Ami azt jelenti a számomra, hogy…

Azt jelenti, hogy igazán rugalmas kódot írhatsz. Olyan kódot, ami tisztább (hatékonyabb, egyszerűbb), olyan kódot, amit nem csak *megírni*, hanem *bővíteni* is sokkal, de sokkal könnyebb, méghozzá olyan módokon, amikre nem is gondoltál, amikor eredetileg megírtad a kódot.

Ez azt jelenti, hogy elmehetsz arra a trópusi vakációra, amíg a munkatársaid frissítik a programot, és lehet, hogy a kollégáidnak nem is lesz szükségük a forráskódodra.

A következő oldalon megtudod, hogyan működik mindez.

Nem tudjuk, hogy veled mi a helyzet, de mi a trópusi vakáció lehetőségét különösen ösztönzőnek találjuk.



Ahhoz, hogy megmutathassuk, hogyan működik a többalakúság, vissza kell mennünk egy kicsit, hogy átismételjük, hogy a szokásos módon hogyan vezethetsz be egy hivatkozást, és hozhatsz létre egy objektumot...

Az objektumok bevezetésének és értékkel való ellátásának 3 lépése

```
1        3  2
Dog myDog = new Dog();
```

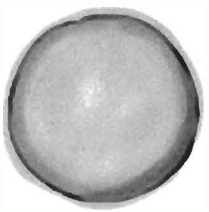
1 Hivatkozó változó bevezetése

`Dog myDog` = `new Dog();`
Ez arra utasítja a JVM-et, hogy foglaljon helyet egy hivatkozás típusú változónak. A hivatkozás örökre Dog típusú lesz. Más szavakkal, egy olyan távirányító, amelyen egy kutya (Dog) – és nem egy macska (Cat), gomb (Button) vagy csatoló (Socket) – vezérlésére szolgáló gombok vannak.



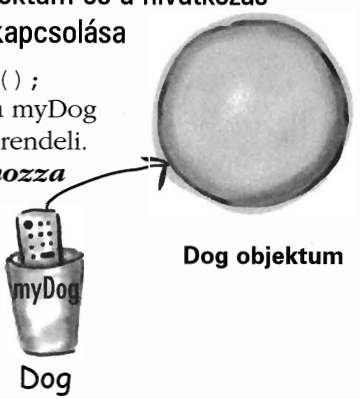
2 Objektum létrehozása

`Dog myDog =` **`new Dog();`**
Ez arra utasítja a JVM-et, hogy foglaljon helyet egy új Dog objektum számára a szemétgyűjtő számára elérhető kupacban.

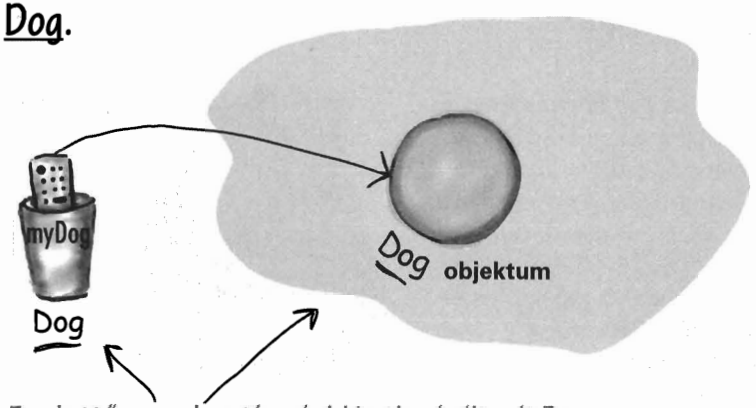


3 Az objektum és a hivatkozás összekapcsolása

`Dog myDog = new Dog();`
Ez az utasítás az új Dog-ot a myDog nevű hivatkozó változóhoz rendeli. Más szavakkal, ***beprogramozza a távirányítót***.



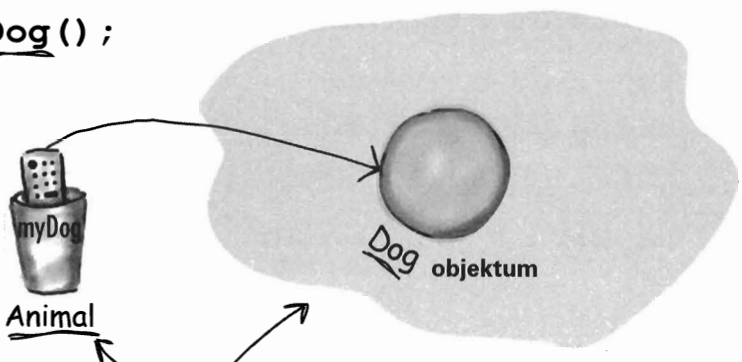
A lényeg az, hogy a hivatkozás típusa
ÉS az objektum típusa ugyanaz.
A példánkban mindkettő Dog.



Ez a kettő ugyanolyan típusú. A hivatkozó változót Dog típusúként vezettük be, az objektumot pedig a new Dog() utasítással hoztuk létre.

A többalakúságnak köszönhetően
azonban a hivatkozás és az objektum
típusa különböző lehet.

```
Animal myDog = new Dog();
```



Ez a kettő NEM azonos típusú. A hivatkozó változót Animal típusúként vezettük be, az objektumot azonban a new Dog() utasítással hoztuk létre.

A többalakúságnak köszönhetően a hivatkozások típusa a tényleges objektumtípus ősosztálya is lehet.



Amikor bevezetsz egy hivatkozó változót, minden objektum, amely teljesíti az IS-A tesztet a változó bejelentett típusára, hozzárendelhető az adott hivatkozáshoz. Más szavakkal, *bármit* hozzárendelhetsz a hivatkozó változóhoz, ami *bővíti* annak a bejelentett típusát. ***Ennek köszönhetően létrehozhatsz például többalakú tömböket.***

Jó, jó, talán egy példa segít.

Bevezetünk egy Animal típusú tömböt, vagyis egy olyan tömböt, amely Animal típusú objektumokat fog tárolni.

```
Animal[] animals = new Animal[5];
animals [0] = new Dog();
animals [1] = new Cat();
animals [2] = new Wolf();
animals [3] = new Hippo();
animals [4] = new Lion();

for (int i = 0; i < animals.length; i++) {

    animals[i].eat();

    animals[i].roam();
}
```

De nézd csak, mit csinálhatsz: az Animal TETSZŐLEGES alosztályát beleteheted az Animal tömbbe!

És ez a többalakúság legjobb része (az egész példa lényege): végigmehetsz a tömbön, meghívhatod az Animal osztály egyik tagfüggvényét, és minden objektum azt csinálja, amit kell!

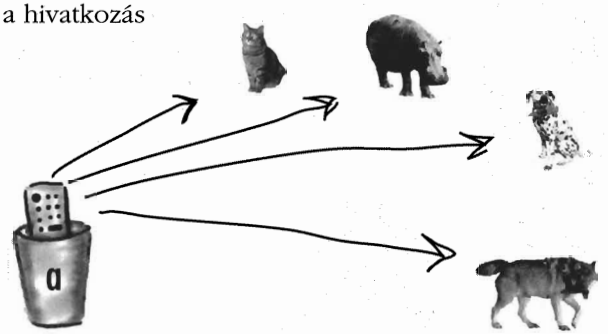
Ha „i" 0, akkor a Dog eat() tagfüggvényét kapod, mert a tömb 0. pozícióján egy Dog áll. Ha „i" 1, a Cat eat() tagfüggvénye lép működésbe.

Ugyanez áll a roam() tagfüggvényre is.

De várj csak! Tovább is van még!

Többalakú argumentumaid és visszatérési típusaid is lehetnek.

Ha bevezethetsz egy őstípusú – legyen ez mondjuk az Animal – hivatkozó változót, és hozzárendelhetsz egy alosztálybeli objektumot – mondjuk egy Dog-ot –, akkor gondolj csak bele, mi történik, ha a hivatkozás egy tagfüggvény argumentuma…




```
class Vet {
    public void giveShot(Animal a) {
        // szörnyű dolgokat teszünk az állattal
        // az "a" paraméter másik végén
        a.makeNoise();
    }
}
```

Az Animal paraméter BÁRMILYEN Animal-altípust elfogad argumentumként. Amikor az állatorvos (Vet) beadta az oltást (giveShot), utasítja az állatot, hogy adjon ki hangot (makeNoise), és annak az állatfajtának a makeNoise() tagfüggvénye fog lefutni, amelyik éppen a kupacban található.

```
class PetOwner {
    public void start() {
        Vet v = new Vet();
        Dog d = new Dog();
        Hippo h = new Hippo();
        v.giveShot(d);
        v.giveShot(h);
    }
}
```

A Vet osztály giveShot() tagfüggvénye bármilyen állatot elfogad. Amíg az argumentumként átadott objektum az Animal alosztálya, a kód működni fog.

A Dog makeNoise() tagfüggvénye fut le

A Hippo makeNoise() tagfüggvénye fut le



A többalakúságnak köszönhetően olyan kódot írhatsz, amit nem kell módosítani, ha új alosztályokat vezetsz be a programban.

Emlékszel a Vet osztályra? Ha úgy írod meg, hogy *Animal* típusúként vezeted be az argumentumokat, akkor a kódod az Animal *bármelyik* alosztályát képes lesz kezelni. Ez azt jelenti, hogy ha mások fel szeretnénk használni a Vet osztályodat, csak annyit kell tenniük, hogy gondoskodnak róla, hogy az *ő* új állattípusaik az Animal osztályt bővítsék. A Vet tagfüggvényei továbbra is működni fognak, pedig amikor a Vet osztályt írtad, mit sem tudtál azokról az új állattípusokról, amelyeket a Vet-nek kezelnie kell majd.

Agytorna

Miért biztos, hogy a többalakúság működik? Miért tételezhetjük fel biztonságosan, hogy minden *alosztály* rendelkezik az *alaposztályból* meghívott tagfüggvénnyel (vagyis azzal az alaposztálybeli hivatkozástípussal, amelyikre a pont műveletet alkalmazod)?

Nincsenek hülye kérdések

K: Van valamilyen gyakorlati korlát az alosztályszintek számára nézve? Milyen mélyre lehet menni?

V: Ha megnézed a Java API-t, láthatod, hogy a legtöbb öröklési hierarchia széles, de nem mély: a legtöbb csupán egy vagy két szintnyi, bár vannak kivételek (különösen a grafikus felületi osztályok között). Rájöhetsz, hogy általában ésszerűbb sekély öröklési fákat meghatározni, de erre nézve nincs szigorú korlát (legalábbis nem olyan, amibe valaha is beleütköznél).

K: Eszembe jutott valami...ha nem férek hozzá egy osztály forráskódjához, de változtatni szeretnék az osztály egyik tagfüggvényének a működésén, megoldhatom ezt alosztályokkal? Bővíthetem a „rossz" osztályt, és felülbírálhatom a tagfüggvényt a saját, jobb kódommal?

V: Aha. Ez az egyik klassz vonása az objektumközpontúságnak, ami megkímélhet attól, hogy a semmiből újra kelljen írnod egy osztályt, vagy fel kelljen kutatnod a programozót, aki elrejtette a forráskódot.

K: *Bármilyen* osztály bővíthető? Vagy az osztályokra is ugyanaz áll, mint az osztálytagokra, vagyis ha egy osztály privát, akkor nem örökölhetünk tőle?

V: Nem létezik olyan, hogy privát osztály, kivéve az úgynevezett *belső* osztályok (inner class) különleges esetét, amivel még nem találkoztunk. Mindazonáltal *van* három dolog, ami megakadályozhatja, hogy egy osztályból alosztályokat származtass.

Az első a hozzáférés-szabályozás (access control). Bár egy osztály *nem* jelölhető meg privátként (`private`), attól még *lehet* nem nyilvános (ami akkor következik be, ha az osztályt nem a `public` kulcsszóval vezeted be). A nem nyilvános osztályokból csak olyan alosztályok származtathatók, amelyek ugyanabban a csomagban találhatók, mint az alaposztály. Más csomagban levő osztályok nem lehetnek egy nem nyilvános osztály alosztályai (és ha már itt tartunk, nem is *használhatják* azt).

A második dolog, ami megakadályozza, hogy alosztályokat származtass egy osztályból, a `final` (végleges) módosító kulcsszó. Ha egy osztály `final`, az azt jelenti, hogy az osztály az öröklési lánc végén áll. Soha senki nem bővíthet egy végleges osztályt.

A harmadik eset az, amikor egy osztálynak csak `private` konstruktorai vannak (a konstruktorokról a 9. fejezetben lesz szó). Ilyen esetben nem származtathatsz belőle alosztályokat.

K: Mikor lehet szükség végleges osztályokra? Milyen előnyei vannak annak, ha megakadályozzuk az alosztályok származtatását egy osztályból?

V: Általában nincs szükség arra, hogy az osztályaid véglegesek legyenek. De ha biztonságra vágysz – annak a biztonságára, hogy tudod, hogy a tagfüggvényeid mindig úgy fognak működni, ahogy megírtad őket (mivel nem lehet őket felülírni) – egy végleges osztály megadhatja neked ezt. A Java API-ban egy csomó osztály ezért végleges. A String osztály például `final`, mert óriási kavarodást okozna, ha valaki egyszer csak megváltoztatná a karakterláncok viselkedését.

K: Akár egy *tagfüggvényt* is tehetek véglegessé, anélkül, hogy az egész *osztály* végleges lenne?

V: Ha egy konkrét tagfüggvény felülírását szeretnéd megakadályozni, jelöld meg a *tagfüggvényt* a `final` módosítóval. A teljes *osztály* akkor legyen `final`, ha azt akarod biztosítani, hogy az osztály *egyik* tagfüggvényét se lehessen felülbírálni.

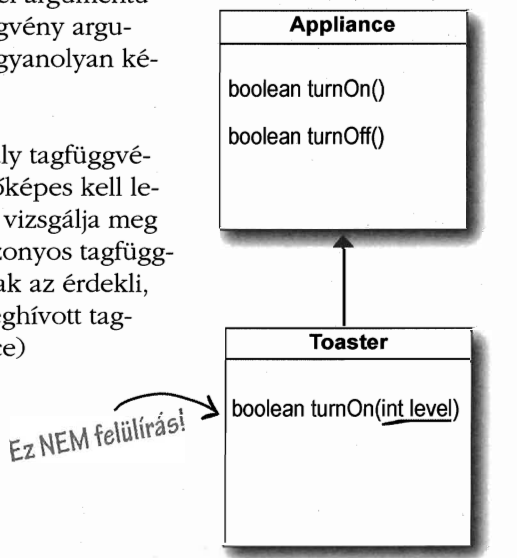
A szerződés betartása: a felülírás szabályai

Amikor felülbírálod egy alaposztály egy tagfüggvényét, beleegyezel, hogy betartod a szerződést. A szerződés azt mondhatja például, hogy „Nem fogadok el argumentumokat, és logikai értéket adok vissza". Más szavakkal, a felülíró tagfüggvény argumentumainak és visszatérési típusainak a külvilág számára *pontosan* ugyanolyan képet kell mutatniuk, mint az alaposztály felülírt tagfüggvényében.

A tagfüggvények *maguk* jelentik a szerződést.

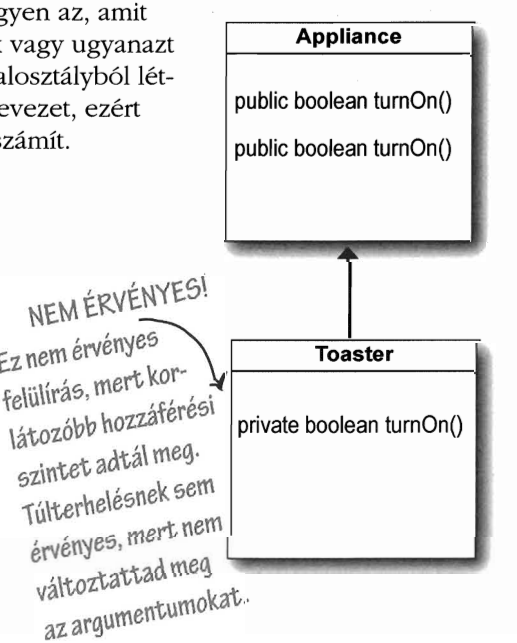
Ahhoz, hogy a többalakúság működjön, az Appliance (Készülék) osztály tagfüggvényének a Toaster (Kenyérpirító) osztályban felülírt változata is működőképes kell legyen futásidőben. Emlékezz vissza, hogy a fordító a hivatkozás típusát vizsgálja meg annak eldöntéséhez, hogy az adott hivatkozásra meghívhatsz-e egy bizonyos tagfüggvényt. Mivel a Toaster-re egy Appliance hivatkozás mutat, a fordítót csak az érdekli, hogy az *Appliance* osztályban megvan-e az Appliance hivatkozásra meghívott tagfüggvény. Futáskor azonban a JVM nem a *hivatkozás* típusát (Appliance) nézi, hanem a kupacban található tényleges *Toaster* objektumot.

Ha tehát a fordítóprogram *jóváhagyta* a tagfüggvényhívást, az csak úgy működhet, ha a felülíró tagfüggvény ugyanazokkal az argumentumokkal és visszatérési típusokkal rendelkezik. Máskülönben előfordulhatna, hogy egy Appliance-re hivatkozó kódrész argumentum nélküliként hívja meg a turnOn() – bekapcs – tagfüggvényt, miközben a Toaster változata egy int értéket vár. Melyik változat hívódik meg futáskor? Nos, az, amelyik az Appliance osztályban található. Más szavakkal, a ***Toaster turnOn(int level) tagfüggvénye nem felülíró függvény***!



① Az argumentumoknak ugyanazoknak, a visszatérési típusoknak pedig összeegyeztethetőnek kell lenniük.

Az alaposztály szerződése megszabja, hogy más kódrészek hogyan használhatnak egy tagfüggvényt. Bármi is legyen az, amit az alaposztály argumentumként elfogad, a tagfüggvényt felülíró alosztálynak ugyanazt kell elfogadnia, és bármi is legyen az, amit az alaposztály visszatérési típusként bevezet, a felülíró tagfüggvénynek vagy ugyanazt a típust kell megadnia, vagy annak egy altípusát. Ne feledd, hogy egy alosztályból létrehozott objektum garantáltan bármit megtehet, amit az alaposztálya bevezet, ezért nyugodtan visszaadhatsz egy altípust ott, ahol a program az őstípusra számít.



② A tagfüggvény nem lehet kevésbé hozzáférhető.

Ez azt jelenti, hogy a hozzáférési szintnek ugyanolyannak vagy barátságosabbnak kell lennie. Nem írhatsz felül például egy nyilvános tagfüggvényt úgy, hogy priváttá teszed. A hívó kód sokkot kapna, mivel olyasmit hív meg, amiről azt *hiszi* (fordításkor), hogy nyilvános tagfüggvény. Ha futáskor a JVM hirtelen rácsapja az ajtót, mert a felülíró változat privát, a hívó szívrohamot kap.

Eddig két hozzáférési szintről tanultál: a privát és a nyilvános szintről. A másik kettőről az üzembe helyezésről szóló fejezetben (Tedd közzé a kódod!), illetve a B függelékben olvashatsz. Van még egy felülírási szabály, ami a kivételkezeléssel kapcsolatos, de arra csak a kivételeket tárgyaló fejezetben (Kockázatos viselkedés) kerítünk sort.

Tagfüggvény túlterhelése

A tagfüggvény-túlterhelés csupán annyit jelent, hogy van két azonos nevű, de eltérő argumentumlistával rendelkező tagfüggvényed. Pont. A túlterhelt tagfüggvényeknek semmi közük a többalakúsághoz!
A túlterhelés révén több változatot készíthetsz egy tagfüggvényből, a hívók kedvéért eltérő argumentumokkal. Például ha van egy tagfüggvényed, ami csak int értékeket fogad el, a hívó kódnak esetleg int-té kell alakítania egy double-t, mielőtt meghívhatná a tagfüggvényt. Ha a tagfüggvényt azonban túlterhelted egy másik változattal, amely double típusú értéket vár, akkor a hívó dolga egyszerűbb. Az objektumok életciklusával foglalkozó fejezetben több példával is találkozhatsz majd, amikor a konstruktorokat tárgyaljuk. Mivel a túlterhelő tagfüggvények nem is próbálnak eleget tenni az alaposztály által meghatározott többalakúsági szerződésnek, a túlterhelt tagfüggvények sokkal rugalmasabbak.

A túlterhelt tagfüggvény csupán egy másik tagfüggvény, aminek történetesen ugyanaz a neve. Semmi köze az öröklődéshez vagy a többalakúsághoz. A túlterhelt tagfüggvény NEM ugyanaz, mint a felülírt tagfüggvény.

1 A visszatérési típusok eltérhetnek.

A visszatérési típusok a túlterhelt tagfüggvényekben szabadon megváltoztathatók, feltéve, hogy az argumentumlisták eltérnek.

2 Nem változtathatod meg CSAK a visszatérési típust.

Ha csak a visszatérési típus különbözik, nem beszélhetünk érvényes *túlterhelésről* – a fordító azt fogja feltételezni, hogy *felülbírálni* próbálod a tagfüggvényt. Ugyanakkor a felülírás sem lesz érvényes, hacsak a visszatérési típus nem altípusa az alaposztályban bejelentett visszatérési típusnak. Egy tagfüggvény túlterheléséhez meg KELL változtatnod az argumentumlistát, bár a visszatérési típust bármire módosíthatod.

3 A hozzáférési szintet bármelyik irányban *módosíthatod.*

Egy tagfüggvényt nyugodtan túlterhelhetsz egy korlátozóbb tagfüggvénnyel, mivel az új tagfüggvényt semmi nem kötelezi arra, hogy eleget tegyen a túlterhelt tagfüggvény szerződésének.

Érvényes példák a tagfüggvény-túlterhelésre:

```
public class Overloads {

   String uniqueID;

   public int addNums(int a, int b) {
      return a + b;
   }

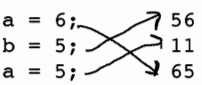
   public double addNums(double a, double b) {
       return a + b;
   }

   public void setUniqueID(String theID) {
      // érvényesség-ellenőrző kód, aztán:
      uniqueID = theID;
   }

   public void setUniqueID(int ssNumber) {
      String numString = "" + ssNumber;
      setUniqueID(numString);
   }
}
```



Összekevert ÜZENETEK



Nézd meg az alábbi rövid Java-programot. A program egyik kódblokkja hiányzik. A feladatod az, hogy összepárosítsd a kódjelölteket (balra) azzal a kimenettel, amit akkor látnál, ha az adott blokkot szúrnád be a kódba. Nem minden kimenetsornak van párja, viszont egyes kimenetsorok többször is felhasználhatók. Kösd össze vonalakkal a kódjelölteket és a nekik megfelelő parancssori kimeneteket!

a program:

```
class A {
  int ivar = 7;
  void m1() {
    System.out.print("A's m1, ");
  }
  void m2() {
    System.out.print("A's m2, ");
  }
  void m3() {
    System.out.print("A's m3, ");
  }
}

class B extends A {
  void  m1() {
    System.out.print("B's m1, ");
  }
}

class C extends B {
  void m3() {
    System.out.print("C's m3, "+(ivar + 6));
  }
}

public class Mixed2 {
  public static void main(String [] args) {
    A a = new A();
    B b = new B();
    C c = new C();
    A a2 = new C();
    
    
  }
}
```

ide jön a kódjelölt (3 sor)

kódjelöltek:

```
b.m1();
c.m2();
a.m3();
```

```
c.m1();
c.m2();
c.m3();
```

```
a.m1();
b.m2();
c.m3();
```

```
a2.m1();
a2.m2();
a2.m3();
```

kimenet:

```
A's m1, A's m2, C's m3, 6

B's m1, A's m2, A's m3,

A's m1, B's m2, A's m3,

B's m1, A's m2, C's m3, 13

B's m1, C's m2, A's m3,

B's m1, A's m2, C's m3, 6

A's m1, A's m2, C's m3, 13
```



Gyakorlat



Légy a Java virtuális gép!

A bal oldali osztályokba beszúrva és lefordítva melyik jobb oldali A-B tagfüggvénypár eredményezné az alul látható kimenetet? (Az A tagfüggvényt a Monster, a B tagfüggvényt a Vampire osztályba kell beszúrni.)

```
public class MonsterTestDrive {
  public static void main(String [] args) {
    Monster [] ma = new Monster[3];
    ma[0] = new Vampire();
    ma[1] = new Dragon();
    ma[2] = new Monster();
    for(int x = 0; x < 3; x++) {
      ma[x].frighten(x);
    }
  }
}

class Monster {
  A
}

class Vampire extends Monster {
  B
}

class Dragon extends Monster {
  boolean frighten(int degree) {
    System.out.println("breath fire");
    return true;
  }
}
```

```
File Edit Window Help SaveYourself
% java MonsterTestDrive
a bite?
breath fire
arrrgh
```

1

A
```
boolean frighten(int d) {
  System.out.println("arrrgh");
  return true;
}
```

B
```
boolean frighten(int x) {
  System.out.println("a bite?");
  return false;
}
```

2

A
```
boolean frighten(int x) {
  System.out.println("arrrgh");
  return true;
}
```

B
```
int frighten(int f) {
  System.out.println("a bite?");
  return 1;
}
```

3

A
```
boolean frighten(int x) {
  System.out.println("arrrgh");
  return false;
}
```

B
```
boolean scare(int x) {
  System.out.println("a bite?");
  return true;
}
```

4

A
```
boolean frighten(int z) {
  System.out.println("arrrgh");
  return true;
}
```

B
```
boolean frighten(byte b) {
  System.out.println("a bite?");
  return true;
}
```

Halászd ki a medencéből!





A ***feladatod*** az, hogy kihalászd a kóddarabokat a medencéből, és elhelyezd azokat a kód üresen hagyott helyein. Egy kóddarabot egynél többször is felhasználhatsz, viszont nem lesz szükséged minden darabra. A ***cél*** olyan osztályok felépítése, amelyek lefordíthatók és együtt, programként futtathatók. Vigyázat – ez a feladat nehezebb, mint gondolnád!

```
public class Rowboat _________ _________ {
   public ____________ rowTheBoat() {
      System.out.print("stroke natasha");
   }
}
```

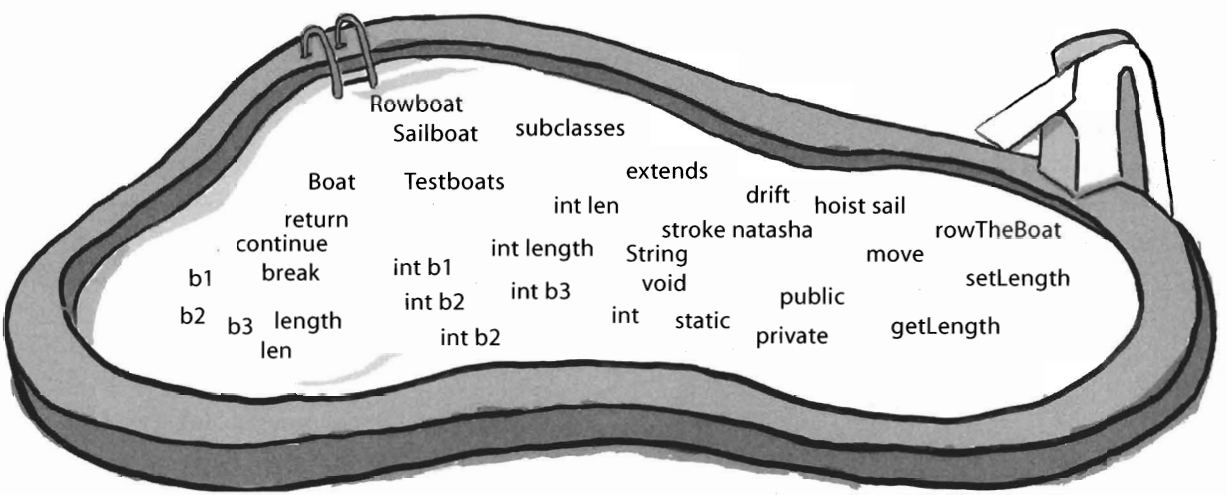
```
public class _________ {
   private int __________ ;
   _______ void _________ ( ______ ) {
      length = len;
   }
   public int getLength() {
      _________ __________ ;
   }
   public ____________ move() {
      System.out.print("____________");
   }
}
```

```
public class TestBoats {
   ______ _______ ________ main(String[] args){
      __________ b1 = new Boat();
      Sailboat b2 = new ___________();
      Rowboat _________ = new Rowboat();
      b2.setLength(32);
      b1.___________();
      b3.___________();
      ________.move();
   }
}
```

```
public class ___________ _________ Boat {
   public _______ _________() {
      System.out.print("____________");
   }
}
```

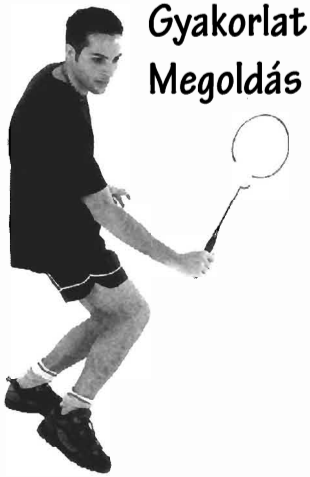
KIMENET:

```
drift  drift  hoist sail
```





Légy a Java virtuális gép!



Gyakorlat Megoldás

Az 1. pár működik.
A 2. pár a Vampire visszatérési típusa (int) miatt nem fordítható le. A Vampire osztály frighten() tagfüggvénye (B) nem érvényes felülírása VAGY túlterhelése a Monster osztály frighten() tagfüggvényének. A visszatérési típus módosítása ÖNMAGÁBAN nem elég az érvényes túlterheléshez, és mivel az int nem egyeztethető össze a boolean típussal, a tagfüggvény felülírásnak sem érvényes. (Emlékezz vissza, hogy ha CSAK a visszatérési típust változtatod meg, akkor a módosított típusnak összeegyeztethetőnek kell lennie az alaposztálybeli változat visszatérési típusával, mely esetben *felülírásról* van szó.)
A 3. és 4. pár lefordítható, de ezt eredményezik:

```
arrrgh

breath fire

arrrgh
```

Vedd észre, hogy a Vampire osztály nem *bírálta* felül a Monster osztály frighten() tagfüggvényét. (A Vampire osztály frighten() tagfüggvénye a 4. párban `byte`, és *nem* `int` értéket vár.)

Összekevert ÜZENETEK

kódjelöltek:

```
b.m1();
c.m2();
a.m3();

c.m1();
c.m2();
c.m3();

a.m1();
b.m2();
c.m3();

a2.m1();
a2.m2();
a2.m3();
```

kimenet:

```
A's m1, A's m2, C's m3, 6
B's m1, A's m2, A's m3,
A's m1, B's m2, A's m3,
B's m1, A's m2, C's m3, 13
B's m1, C's m2, A's m3,
B's m1, A's m2, C's m3, 6
A's m1, A's m2, C's m3, 13
```



```
public class Rowboat extends Boat {
   public void rowTheBoat() {
      System.out.print("stroke natasha");
   }
}
public class Boat {
   private int length ;
   public void setLength ( int len ) {
      length = len;
   }
   public int getLength() {
      return length ;
   }
   public void move() {
      System.out.print("drift ");
   }
}
```

```
public class TestBoats {
   public static void main(String[] args){
      Boat b1 = new Boat();
      Sailboat b2 = new Sailboat();
      Rowboat b3 = new Rowboat();
      b2.setLength(32);
      b1.move();
      b3.move();
      b2.move();
   }
}
public class Sailboat extends Boat {
   public void move() {
      System.out.print("hoist sail ");
   }
}
```

KIMENET:

```
drift  drift  hoist sail
```

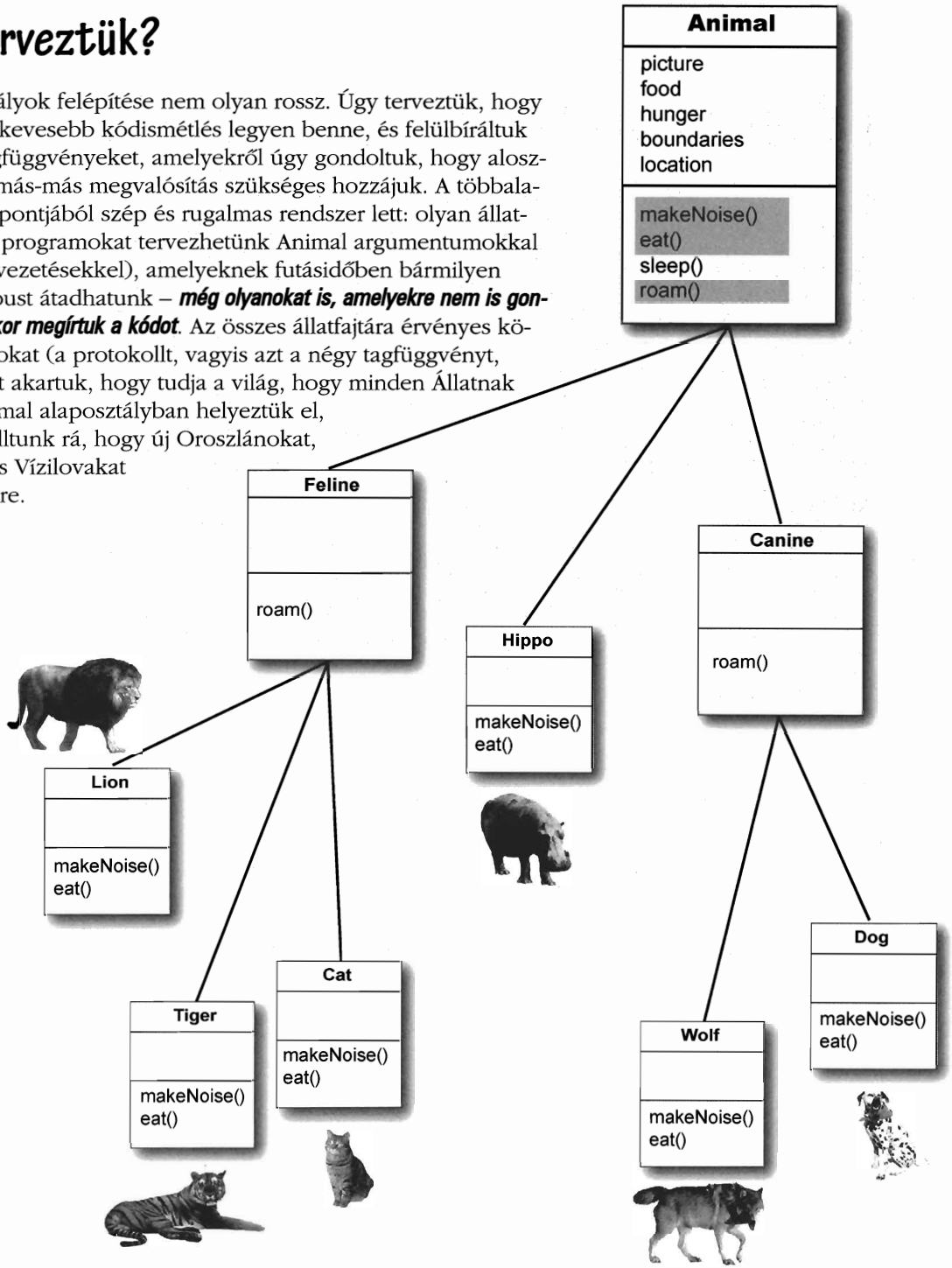
Komoly többalakúság



Az öröklés csak a kezdet. A többalakúság előnyeinek kiaknázásához felületekre (interface) van szükséged (nem GUI-félékre). Túl kell lépned az egyszerű öröklésen, hogy olyan rugalmasságot és bővíthetőséget érj el, ami csak felületek tervezése és felületekre építő kódolás révén lehetséges. A Java legklasszabb lehetőségeinek egy része nem is működne felületek nélkül, így ha nem is magad tervezed, akkor is használnod kell majd őket. Persze *akarsz* majd felületeket tervezni. *Szükséged lesz* rá, hogy felületeket tervezz. ***Nem fogod érteni, hogyan éltél eddig nélkülük.*** Mik is a felületek? 100 %-ig elvont osztályok. Mi az, hogy elvont osztály? Olyan osztály, amelyből nem lehet példányokat létrehozni. Az meg mire jó? Várj egy kicsit, és megtudod. De ha visszaidézed az előző fejezet végét, ahol többalakú argumentumokat használtunk, hogy egyetlen Vet-tagfüggvény mindenféle típusú Animal-alosztályt fogadhasson – nos, ott már megkarcoltuk a felszínt. A felületek jelentik a ***polit*** a polimorfizmusban. Az ***ab***-t az absztraktban. A ***koffeint*** a Javában.

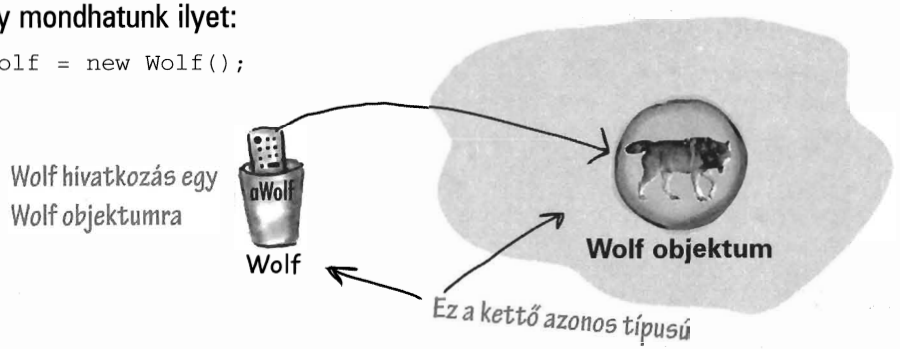
Elfelejtettünk valamit, amikor ezt megterveztük?

Az állatosztályok felépítése nem olyan rossz. Úgy terveztük, hogy a lehető legkevesebb kódismétlés legyen benne, és felülbíráltuk azokat a tagfüggvényeket, amelyekről úgy gondoltuk, hogy alosztályonként más-más megvalósítás szükséges hozzájuk. A többalakúság szempontjából szép és rugalmas rendszer lett: olyan állatfelhasználó programokat tervezhetünk Animal argumentumokkal (és tömbbevezetésekkel), amelyeknek futásidőben bármilyen Animal-altípust átadhatunk – ***még olyanokat is, amelyekre nem is gondoltunk, amikor megírtuk a kódot***. Az összes állatfajtára érvényes közös szabályokat (a protokollt, vagyis azt a négy tagfüggvényt, amelyről azt akartuk, hogy tudja a világ, hogy minden Állatnak van) az Animal alaposztályban helyeztük el, és készen álltunk rá, hogy új Oroszlánokat, Tigriseket és Vízilovakat hozzunk létre.



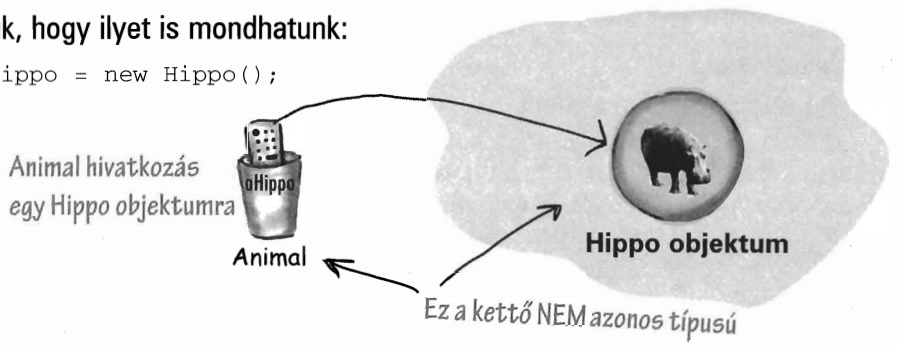
Tudjuk, hogy mondhatunk ilyet:

```
Wolf aWolf = new Wolf();
```



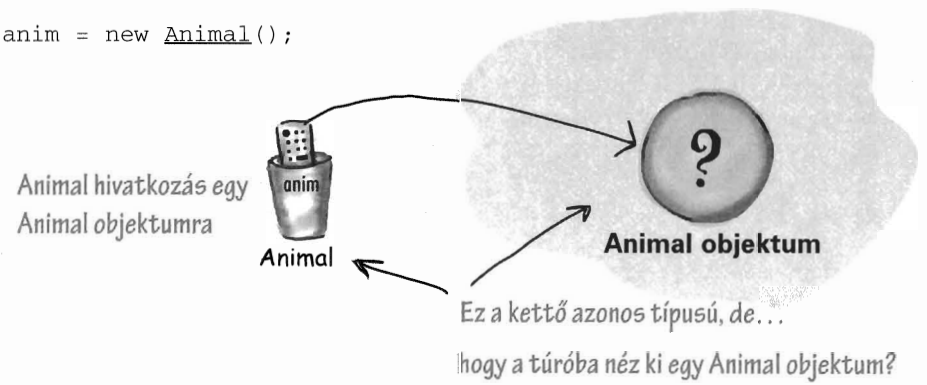
És azt is tudjuk, hogy ilyet is mondhatunk:

```
Animal aHippo = new Hippo();
```

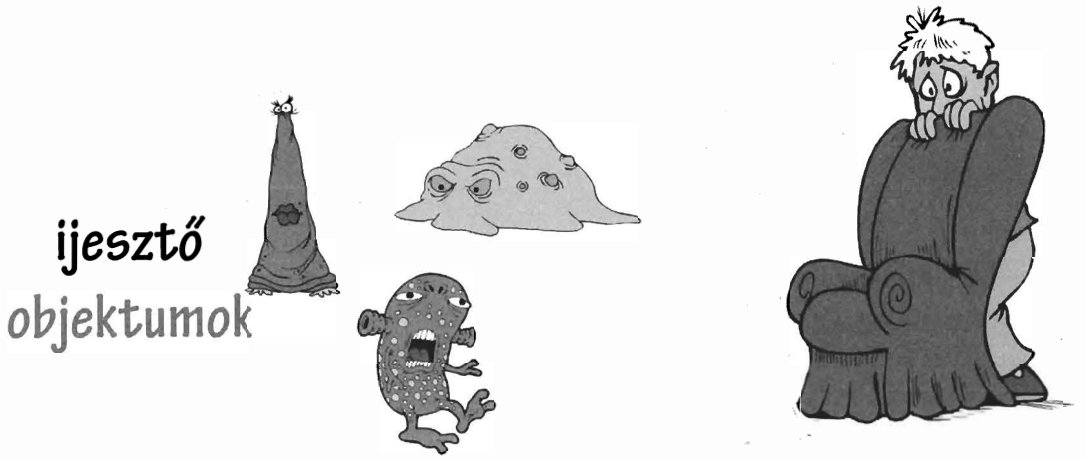


A dolog itt kezd bizarrá válni:

```
Animal anim = new Animal();
```



Hogyan *néz ki* egy új Animal() objektum?



Mi az értéke a példányváltozóknak? Bizonyos osztályokat nem szabad példányosítani!

Wolf (Farkas), Hippo (Víziló) vagy Tiger (Tigris) objektumot van értelme létrehozni, de az Animal (Állat) objektum pontosan *micsoda*? Milyen alakja van? Milyen a színe vagy a mérete? Hány lába van?
Egy Animal típusú objektum létrehozásával próbálkozni olyan rémálom, mint egy **transzporterbaleset a Star Trekben™**. Tudod, amikor a „sugározz fel" során valami csúnya hiba történik.
De hogyan oldhatjuk meg a problémát? Az öröklés és a többalakúság miatt *szükség* van az Animal osztályra, viszont azt szeretnénk, hogy a programozók csak az Animal kevésbé elvont *alosztályait* példányosíthassák, nem magát az Animal osztályt. Tiger és Lion objektumokat akarunk, ***nem Animal objektumokat***.
Szerencsére van egy egyszerű módja annak, hogy megakadályozd egy osztály példányosítását, vagyis azt, hogy bárki kiadhasson egy „**new**" utasítást az adott típusra. Ha az osztályt az **abstract** (elvont) jelöléssel látod el, a fordítóprogram minden kódot (bárhol legyenek is) megakadályoz abban, hogy példányokat hozzanak létre az adott típusból.
Ezt az elvont típust továbbra is használhatod hivatkozástípusként – végtére is, ez az egyik fő oka annak, amiért az elvont osztály egyáltalán létezik (hogy többalakú argumentumként vagy visszatérési típusként használhasd, vagy hogy többalakú tömböket hozhass létre).
Amikor megtervezed az osztályaid öröklési rendszerét, el kell döntened, hogy mely osztályok legyenek *elvontak*, és melyek *konkrétak*. A konkrét osztályok azok, amelyek elég konkrétak ahhoz, hogy példányokat hozhass létre belőlük. Az, hogy egy osztály *konkrét*, csupán annyit jelent, hogy nyugodtan létrehozhatsz ilyen típusú objektumokat.
Egy osztály elvonttá tétele könnyű: csak ki kell tenned az **abstract** kulcsszót az osztály bevezetése elé:

```
abstract class Canine extends Animal {
   public void roam() { }
}
```

A fordító nem engedi, hogy példányosíts egy elvont osztályt

Ha egy osztály elvont, az azt jelenti, hogy soha senki nem hozhat létre belőle új példányt. Az elvont osztályt továbbra is használhatod, mint a bejelentett hivatkozástípust, többalakúság céljára, de nem kell attól tartanod, hogy valaki ilyen típusú objektumokat hoz létre – ezt a fordítóprogram *garantálja*.

```
abstract public class Canine extends Animal
{

   public void roam() { }

}
```

```
public class MakeCanine {

   public void go() {

      Canine c;
      c = new Dog();
      c = new Canine();
      c.roam();

   }

}
```

Ez oké, mert egy alosztálybeli objektumot mindig hozzárendelhetsz egy az alaposztályra mutató hivatkozáshoz, akkor is, ha az alaposztály elvont.

A Canine osztály elvont megjelölésű, ezért a fordító ezt NEM engedi meg.

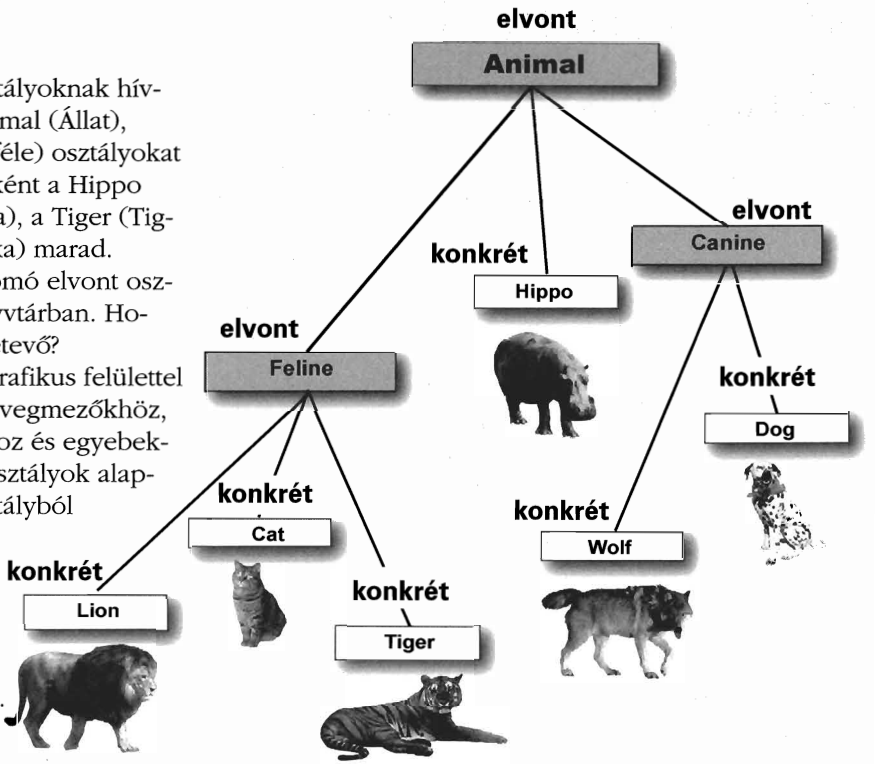
```
File Edit Window Help BeamMeUp
% javac MakeCanine.java
MakeCanine.java:5: Canine is abstract;
cannot be instantiated
      c = new Canine();
          ^
1 error
```

Az **elvont osztályoknak** lényegében semmi hasznuk*, semmi értékük, semmi céljuk az életben, hacsak nem ***bővíted*** őket.
Ha egy osztály elvont, akkor futásidőben az elvont osztály ***alosztályainak* példányai** végzik a munkát.

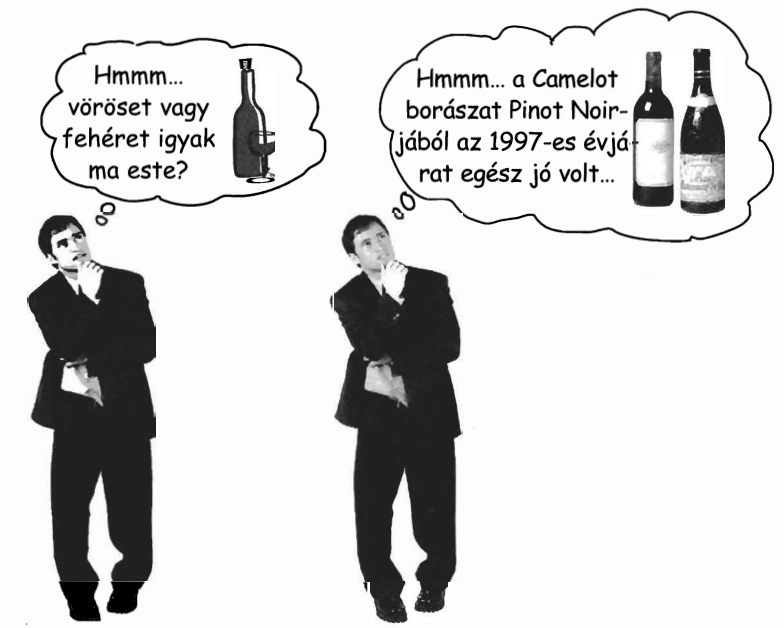
*Ez alól van egy kivétel: egy elvont osztálynak lehetnek statikus tagjai (lásd a 10. fejezetet).

Elvont vagy konkrét?

A nem elvont osztályokat *konkrét* osztályoknak hívjuk. Az állatok öröklési fájában az Animal (Állat), Canine (Kutyaféle) és Feline (Macskaféle) osztályokat tehetjük elvonttá, így konkrét osztályként a Hippo (Víziló), a Wolf (Farkas), a Dog (Kutya), a Tiger (Tigris), a Lion (Oroszlán) és a Cat (Macska) marad.
Ha végiglapozod a Java API-t, egy csomó elvont osztályt találhatsz, különösen a GUI könyvtárban. Hogyan néz ki egy grafikus felületi összetevő?
A Component (Összetevő) osztály a grafikus felülettel (GUI) kapcsolatos, a gombokhoz, szövegmezőkhöz, gördítősávokhoz, párbeszédablakokhoz és egyebekhez hasonló elemeket meghatározó osztályok alaposztálya. Az általános *Component* osztályból nem hozunk létre példányokat, hogy a képernyőre helyezve mondjuk JButton gombokat jelenítsünk meg. Más szavakkal, csak a Component *konkrét alosztályait* példányosítjuk, sohasem magát a Component osztályt.



Agytorna

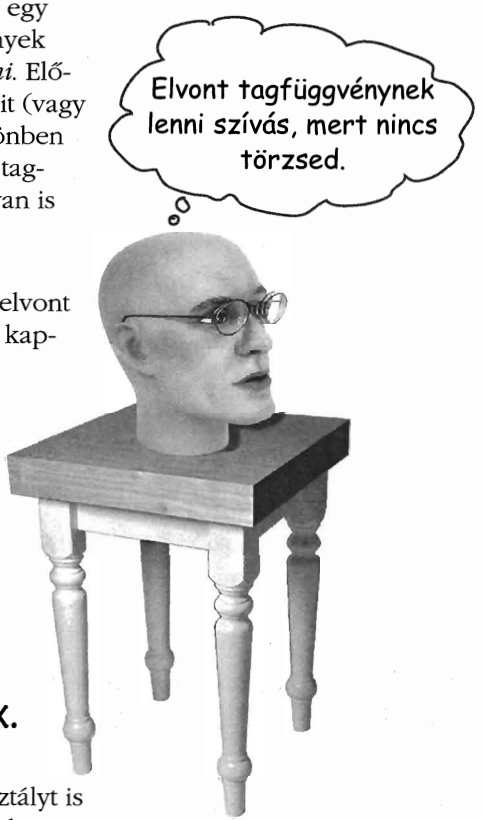


elvont vagy konkrét?

Honnan tudhatod, hogy egy osztálynak elvontnak kell-e lennie? A ***Wine*** (Bor) valószínűleg elvont. De mi a helyzet a ***Red*** (Vörös) és ***White*** (Fehér) osztályokkal? Valószínűleg ezek is elvontak (legalábbis némelyeknek). De akkor mikor válnak a dolgok konkréttá a hierarchiában?
A ***PinotNoir*** már lehet konkrét, vagy ennek is elvontnak kell lennie? A Camelot Vineyards 1997 Pinot Noir mindenesetre biztosan konkrét lesz. De honnan tudhatjuk biztosan?
Nézd meg az állatok fenti öröklési fáját: helyes döntéseket hoztunk azt illetően, hogy mely osztályok legyenek elvontak, és melyek konkrétak? Változtatnál bármit is az Animal öröklési fán (természetesen újabb állattípusok felvételén kívül)?

Elvont tagfüggvények

Az osztályok mellett a tagfüggvényeket is megjelölheted elvontként. Ha egy osztály elvont, az azt jelenti, hogy az osztályt *bővíteni* kell; a tagfüggvények esetében az elvontság jelentése az, hogy a tagfüggvényt *felül kell bírálni*. Előfordulhat, hogy úgy döntesz, hogy egy elvont osztály egyes viselkedéseit (vagy az összeset) egy konkrétabb alosztálynak kell megvalósítania, mert különben nincs értelmük. Más szavakkal, nem tudsz kiötleni semmilyen általános tagfüggvény-megvalósítást, ami hasznára lehetne az alosztályoknak – hogyan is nézne ki egy általános *eat()* – eszik – tagfüggvény?



Az elvont tagfüggvényeknek nincs törzsük!

Mivel már eldöntötted, hogy nincs olyan kód, aminek értelme lenne az elvont tagfüggvényben, nem teszel semmit a tagfüggvény törzsébe. Nincsenek kapcsos zárójelek – a bevezetést egyszerűen lezárod egy pontosvesszővel:

```
public abstract void eat();
```

Nincs függvénytörzs! Zárd le pontosvesszővel!

Ha elvont tagfüggvényt vezetsz be, az osztályt is KÖTELEZŐ elvontként megjelölnöd. Egy nem elvont osztályban nem lehetnek elvont tagfüggvények.

Ha egy osztályba akár csak egyetlen elvont tagfüggvényt is teszel, az osztályt is elvonttá kell tenned. Az elvont osztályokban azonban *megteheted*, hogy kevered az elvont és nem elvont tagfüggvényeket.

Nincsenek hülye kérdések

K: Mi a *célja* az elvont tagfüggvényeknek? Azt hittem, pont azért hozunk létre elvont osztályokat, hogy olyan közös kódot helyezzünk el bennük, amit az alosztályok örökölhetnek.

V: Örökölhető tagfüggvény-megvalósításokat (vagyis valódi *törzzsel* rendelkező tagfüggvényeket) tenni egy alaposztályba Jó Dolog. *Amikor van értelme.* Egy elvont osztályban azonban sokszor *nincs*, mert nem lehet olyan általános kódot írni, amit az alosztályok hasznosnak találnának. Az elvont tagfüggvények célja az, hogy bár nem adsz meg semmilyen tényleges függvénykódot, az altípusok (alosztályok) számára részben meghatározod a *protokollt*.

K: Ami azért jó, mert...

V: Többalakúság! Emlékezz vissza, hogy a cél az, hogy az alaposztály típusát (ami gyakran elvont) tagfüggvény-argumentumként, visszatérési típusként vagy tömbtípusként használhasd. Így anélkül adhatsz új altípusokat (például új Animal-alosztályokat) a programhoz, hogy át kellene írnod a tagfüggvényeket, vagy újakat kellene készítened az új típusok kezelésére. Gondolj csak bele, hogyan kellene módosítanod a Vet osztályt, ha a tagfüggvények számára nem az Animal-t adná meg argumentumtípusként. Minden egyes Animal-alosztályhoz külön tagfüggvényt kellene írnod: egyet, ami egy Lion-t vár, egyet, ami egy Wolf-ot, és így tovább, érted a lényeget. Egy elvont tagfüggvénnyel azonban azt mondod, hogy „Ennek a típusnak minden altípusa rendelkezik EZZEL a tagfüggvénnyel", amivel elősegíted a többalakúságot.

Minden elvont tagfüggvényt meg KELL valósítanod



Egy elvont tagfüggvényt *megvalósítani* pont olyan, mint *felülbírálni* egy tagfüggvényt.

Az elvont tagfüggvényeknek nincs törzsük: kizárólag a többalakúság biztosítása végett léteznek. Ez azt jelenti, hogy az öröklési fában az első konkrét osztálynak *minden* elvont tagfüggvényt kötelező megvalósítania.

A feladatot azonban átpasszolhatod, ha magad is elvont vagy. Például ha az Animal és a Canine osztály egyaránt elvont, és mindkettőnek vannak elvont tagfüggvényei, a Canine osztálynak nem kell megvalósítania az Animal elvont tagfüggvényeit. Amint azonban eljutunk az első konkrét alosztályhoz, például a Dog-hoz, annak már meg kell valósítania minden elvont tagfüggvényt, ami az Animal *és* a Canine osztályban szerepel.

Ugyanakkor jusson eszedbe, hogy egy elvont osztálynak elvont és *nem* elvont tagfüggvényei is lehetnek, így a Canine osztály például megvalósíthat egy elvont tagfüggvényt az Animal osztályból, hogy a Dog-nak már ne kelljen. Ha azonban a Canine nem mond semmit az Animal elvont tagfüggvényeiről, akkor a Dog-nak az Animal valamennyi elvont tagfüggvényét meg kell valósítania.

Amikor azt mondjuk, hogy „meg kell valósítanod az elvont tagfüggvényt", az alatt azt értjük, hogy *meg kell adnod egy törzset a számára.* Ez annyit tesz, hogy létre kell hoznod egy nem elvont tagfüggvényt az osztályban, azonos tagfüggvény-aláírással (signature; ez a tagfüggvény nevét és argumentumait jelenti), illetve olyan visszatérési típussal, amely összeegyeztethető az elvont tagfüggvény bejelentett visszatérési típusával. Az, hogy *mit* teszel a tagfüggvénybe, már tőled függ. A Javát csak az érdekli, hogy a tagfüggvény *ott legyen* a konkrét alosztályban.



Hegyezd ki a ceruzád!

Elvont és konkrét osztályok

Használd fel konkrétan az imént szerzett elvont tudást! A középső oszlopban néhány osztályt soroltunk fel. A feladatod az, hogy kigondold, milyen alkalmazásokban lehetne az adott osztály konkrét, és milyen alkalmazásokban elvont. Pár példát beírtunk helyetted, hogy megadjuk a kezdőlökést. A Fa osztály például elvont lenne egy faiskola-programban, ahol számít a Tölgy és a Rezgőnyárfa közötti különbség, egy golfszimulációs programban azonban a Fa konkrét osztály lehet (esetleg az Akadály alosztálya), mivel ez utóbbi program nem különbözteti meg a fafajtákat. (Természetesen nem csak egyféle helyes megoldás létezik – minden az adott program szerkezetétől függ.)

Konkrét	**Osztály**	**Elvont**
golfszimulációs program	Fa	faiskola-alkalmazás
_____	Ház	építészeti program
műholdfotó-alkalmazás	Város	_____
_____	Futballista	edzőprogram
_____	Szék	_____
_____	Vásárló	_____
_____	Rendelés	_____
_____	Könyv	_____
_____	Bolt	_____
_____	Beszállító	_____
_____	Golfklub	_____
_____	Karburátor	_____
_____	Sütő	_____

A többalakúság működés közben

Tegyük fel, hogy *saját* listaosztályt szeretnél írni. Olyat, amely Dog objektumokat tárolhat, de tételezzük fel egy pillanatra, hogy nincs tudomásod az ArrayList osztályról. Az első körben csak egy *add()* tagfüggvényt adsz hozzá. A hozzáadott Dog objektumokat egy egyszerű Dog tömbben (Dog[]) tárolod, amelynek a hosszát 5-ben szabod meg. Ha a tömb elérte az 5 Dog objektumban megszabott korlátot, továbbra is meghívhatod az *add()* tagfüggvényt, de az nem fog csinálni semmit. Ha a korlátot még *nem* érted el, az *add()* tagfüggvény elhelyezi a Dog-ot a tömb következő elérhető rekeszében, majd növeli a következő elérhető rekeszt jelölő sorszámot (nextIndex).

Saját kutyalista építése

(Talán az első kísérlet a világon egy saját ArrayList-szerű osztály készítésére a semmiből)




```
public class MyDogList {

    private Dog [] dogs = new Dog[5];

    private int nextIndex = 0;


    public void add(Dog d) {

        if (nextIndex < dogs.length) {

            dogs[nextIndex] = d;

            System.out.println("Dog added at " + nextIndex);

            nextIndex++;

        }
    }
}
```

A színfalak mögött egy régi, egyszerű Dog tömböt használunk.

Ezt minden alkalommal növeljük, amikor hozzáadunk egy új Kutyát a tömbhöz.

Ha még nem értük el a kutyatömb korlátját, a Dog hozzáadódik a tömbhöz, és kiírunk egy üzenetet.

Növeljük, hogy megkapjuk a következő sorszámot

Ó, jaj, most már macskákat is kell tartanom

Ebben az esetben a következő lehetőségeid vannak:

1) Létrehozhatsz egy külön osztályt – mondjuk MyCatList néven – a Cat objektumok tárolására. Elég esetlen megoldás.
2) Létrehozhatsz egyetlen osztályt (DogAndCatList), amely két különböző tömböt tartalmaz példányváltozókként, és két különböző add() tagfüggvénye van: addCat(Cat c) és addDog(Dog d). Ez is esetlen megoldás.
3) Létrehozhatsz egy sokoldalú AnimalList osztályt, amely *bármilyen* Animal-alosztályt elfogad (mivel azt már tudod, hogy a követelményeket kiegészítették a Macskák hozzáadásával, előbb-utóbb biztosan tudnod kell majd *másféle* állatokat is a programhoz adni). Nekünk ez a megoldás tetszik a legjobban, úgyhogy az osztályunkat általánosabbá tettük, hogy Kutyák helyett Állatokat fogadjon el. A lényeges változtatásokat kiemeltük (a logika természetesen ugyanaz, csak a típus változott Dog-ról Animal-ra mindenhol a kódban).

Saját állatlista építése

2 változat

MyAnimalList

Animal[] animals
int nextIndex

add(**Animal** a)

```
public class MyAnimalList {

    private Animal[] animals = new Animal[5];
    private int nextIndex = 0;

    public void add(Animal a) {
       if (nextIndex < animals.length) {
          animals[nextIndex] = a;
          System.out.println("Animal added at " + nextIndex);
          nextIndex++;

       }
    }
}
```

Ne ess pánikba: nem új Animal objektumot hozunk létre, hanem egy új tömbobjektumot, ami Animal típusú. (Ne feledd, hogy elvont típusból nem hozhatsz létre új példányt, olyan tömbobjektumot azonban LÉTREHOZHATSZ, ami ezt a típust TÁROLJA.)

```
public class AnimalTestDrive{
   public static void main (String[] args) {
      MyAnimalList list = new MyAnimalList();
      Dog a = new Dog();
      Cat c = new Cat();
      list.add(a);
      list.add(c);
   }
}
```

```
File Edit Window Help Harm
% java AnimalTestDrive
Animal added at 0
Animal added at 1
```

Mi a helyzet azokkal az objektumokkal, amelyek nem állatok? Miért nem készítünk egy olyan osztályt, ami elég általános ahhoz, hogy bármit elfogadjon?

Tudod, mire megy ki a játék. A tömb típusát az *add()* tagfüggvény argumentumával együtt valami olyasmire szeretnénk változtatni, ami az Animal *felett* áll. Valami olyasmire, ami még az Animal-nál is *általánosabb* és *elvontabb*. De hogyan? Az Animal-nak *nincs* ősosztálya. Vagy mégis?

Emlékszel az ArrayList tagfüggvényeire? Nézd csak meg oldalt, hogy a remove, contains és indexOf tagfüggvények mind-mind az ***Object*** objektumtípust használják!

3 változat

(Ez csak néhány az ArrayList tagfüggvényei közül – még egy csomó van.)

ArrayList

boolean remove(Object *elem*)
Eltávolítja a paraméterben meghatározott objektum egy példányát. „True"-t ad vissza, ha az elem megtalálható a listában.

boolean contains(Object *elem*)
„True"-t ad vissza, ha az objektum paraméter találatot eredményez.

boolean isEmpty()
„True"-t ad vissza, ha a listában nincsenek elemek.

int indexOf(Object *elem*)
Vagy az objektum paraméter sorszámát (indexét), vagy –1-et ad vissza.

Object get(int *sorszám*)
A megadott sorszámú elemet adja vissza a listából.

boolean add(Object *elem*)
Az elemet a listához adja („true"-t ad vissza).

// satöbbi

Az ArrayList számos tagfüggvénye a végső többalakú típust, az Object-et használja. Mivel a Javában minden osztály az Object alosztálya, ezek az ArrayList-tagfüggvények bármit elfogadnak.

(Megjegyzés: a Java 5.0 óta a get() és add() tagfüggvények valójában kicsit másképp néznek ki, mint amit itt bemutattunk, de egyelőre így jegyezd meg őket. Kicsit később elmondjuk a teljes történetet.)

A Javában minden osztály az Object osztályt bővíti.

Az Object osztály minden osztály anyja: *mindennek* ez az alaposztálya.

Még ha élsz is a többalakúság előnyeivel, akkor is létre kell hoznod egy olyan osztályt, amelynek a tagfüggvényei az *általad megadott* többalakú típust várják és adják vissza. Ha a Javában nem lenne mindennek egy közös alaposztálya, a Java fejlesztői semmiféleképpen nem hozhatnának létre olyan tagfüggvényekkel rendelkező osztályokat, amelyek a *te* egyéni típusaiddal működnek – *vagyis olyan típusokkal, amelyeknek a létezéséről nem is tudtak, amikor megírták az ArrayList osztályt.*

Tehát kezdettől fogva az Object osztály alosztályait hoztad létre, anélkül, hogy tudtál volna róla. ***Minden osztály, amit írsz, az Object bővítése***, és ezt soha nem kell külön jelezned. Viszont gondolhatsz rá úgy, mintha az általad írt osztályok így néznének ki:

```
public class Dog extends Object { }
```

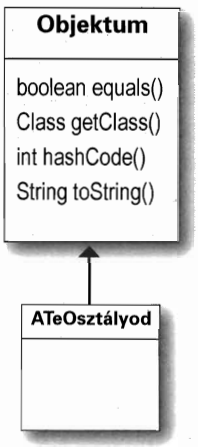
De várjunk csak! A Dog *már* bővít valamit: a *Canine* osztályt! Sebaj, a fordító gondoskodik róla, hogy a *Canine* az Object bővítése legyen. Csakhogy a Canine az Animal osztályt bővíti! Ez sem gond: a fordító elintézi, hogy az *Animal* az Object-et bővítse.

Minden osztály, amely *kifejezetten* nem másik osztályt bővít, *közvetetten* az Object osztályt bővíti.

Tehát, mivel a Dog a Canine bővítése, nem *közvetlenül* bővíti az Object-et (közvetetten viszont igen), és ugyanez áll a Canine osztályra is – az Animal osztály azonban *közvetlen* bővítése az Object-nek.

Mi is *ez az* Object nevű ultra-szuper-megaosztály?

Ha te lennél a Java, milyen viselkedéssel szeretnél *minden* objektumot felruházni? Hm, lássuk csak…esetleg egy olyan tagfüggvénnyel, amelynek a segítségével kideríthető, hogy egy objektum egyenlő-e egy másik objektummal? Vagy egy olyan tagfüggvénnyel, amely meg tudja mondani az említett objektum tényleges osztálytípusát? Vagy mondjuk egy olyan tagfüggvénnyel, amely hasítókódot rendel az objektumhoz, hogy az objektumot hasítótáblákban használhasd? (A Java hasítótábláiról a 17. fejezetben és a B függelékben lesz szó.) Ó, van még jobb: egy olyan tagfüggvénnyel, amely szöveget ír ki az objektumhoz? Tudod, mit? Varázslatos módon az Object osztály mind a négy említett dologhoz biztosít tagfüggvényt. Persze vannak még más tagfüggvényei is, de nekünk ezek érdekesek igazán.



Csak NÉHÁNY az Object osztály tagfüggvényei közül

Minden általad írt osztály örökli az Object osztály összes tagfüggvényét. Tehát az eddig elkészített osztályaid olyan tagfüggvényeket örököltek, amelyekről nem is tudtál.

1 equals(Object o)

```
Dog a = new Dog();
Cat c = new Cat();

if (a.equals(c)) {
    System.out.println("true");
} else {
    System.out.println("false");
}
```

```
File Edit Window Help Stop
% java TestObject
false
```

Azt árulja el neked, hogy két objektum „egyenlőnek" minősül-e (az „egyenlőség" jelentését a B függelékben tisztázzuk).

2 getClass()

```
Cat c = new Cat();
System.out.println(c.getClass());
```

```
File Edit Window Help Faint
% java TestObject
class Cat
```

Azt az osztályt adja vissza, amelyikből az objektumot példányosították.

3 hashCode()

```
Cat c = new Cat();
System.out.println(c.hashCode());
```

```
File Edit Window Help Drop
% java TestObject
8202111
```

Az objektum hasítókódját írja ki (egyelőre gondolj rá úgy, mint egy egyedi azonosítóra).

4 toString()

```
Cat c = new Cat();
System.out.println(c.toString());
```

```
File Edit Window Help LapseIntoComa
% java TestObject
Cat@7d277f
```

Egy karakterlánc formájú üzenetet ír ki az osztály nevével, illetve egy számmal, amivel általában nem foglalkozunk.

Nincsenek hülye kérdések

K: Az Object elvont osztály?

V: Nem, legalábbis a Java hivatalos értelmezése szerint nem az.
Az Object azért nem elvont osztály, mert van benne olyan tagfüggvény-megvalósító kód, amelyet alapértelmezés szerint minden osztály örökölhet és használhat, anélkül, hogy felül kellene bírálnia.

K: Akkor az Object tagfüggvényeit felül *lehet* bírálni?

V: Egyeseket igen. Néhány tagfüggvény azonban `final` megjelölésű, ami azt jelenti, hogy nem írhatók felül. A saját osztályaidban (melegen) ajánlott felülírni a hashCode(), equals() és toString() tagfüggvényeket (ennek módját egy kicsit később tanulhatod meg), a getClass()-hoz hasonló tagfüggvények azonban olyan dolgokat csinálnak, amelyeknek szavatoltan egy bizonyos módon kell működniük.

K: Ha az ArrayList tagfüggvényei elég általánosak ahhoz, hogy az Object típust használják, akkor mit jelent az, hogy ArrayList<DotCom>? Azt hittem, DotCom objektumok tárolására korlátoztam az ArrayList-et.

V: *Valóban* korlátoztad az osztályt.
A Java 5.0 előtt a tömblistákat nem lehetett korlátozni. Lényegében olyanok voltak, mint amikor a Java 5.0-ban azt írod, hogy ArrayList<Object>. Más szavakkal, ha egy tömblistát bármi olyasmire korlátozol, ami Object – ami a Javában minden objektumra igaz –, akkor a tömblista *bármilyen* osztálytípusból példányosítható.
Az új <*típus*> utasítással kapcsolatos részleteket a könyv későbbi részében tárgyaljuk.

K: Jó, akkor térjünk vissza ahhoz, hogy az Object nem elvont (vagyis bizonyára konkrét) osztály. HOGYAN engedhetem valakinek, hogy létrehozzon egy Object típusú objektumot? Ez nem ugyanolyan furcsa, mint egy Animal objektumot létrehozni?

V: Jó kérdés! Hogy miért elfogadható új Object-példányokat létrehozni? Nos, azért, mert néha éppen egy általános objektumot szeretnénk objektumként használni. Egy *pehelysúlyú* objektumot. Az Object típusú objektumokat messze a leggyakrabban szálak összehangolására szokták használni (erről a 15. fejezetben lesz szó). Ezt az információt egyelőre a fiókba teheted; most maradjunk annyiban, hogy ritkán fogsz Object típusú objektumokat létrehozni, annak ellenére, hogy *megteheted.*

K: Akkor mondhatjuk azt, hogy az Object típus főként arra való, hogy többalakú argumentumként és visszatérési típusként használjuk? Mint az ArrayList-ben?

V: Az Object osztálynak *két* fő célja van: egyrészt többalakú típusként viselkedik azokban a tagfüggvényekben, amelyeknek bármely – általad vagy mások által létrehozott – osztállyal működniük kell, másrészt pedig olyan *valódi* tagfüggvénykódot biztosít, amelyre futásidőben minden Java-objektumnak szüksége van. (Az pedig, hogy az Object osztályban kaptak helyet, azt jelenti, hogy minden osztály örökölheti őket.) Az Object legfontosabb tagfüggvényeinek némelyike a szálakkal kapcsolatos; ezekkel a könyv során még találkozni fogsz.

K: Ha a többalakú típusok használata ennyire hasznos, akkor miért nem írunk meg MINDEN tagfüggvényt úgy, hogy Object típusú objektumokat várjon és adjon vissza?

V: Ó, jaj...Gondolj csak bele, mi történne! Először is, szembe mennénk a „típusbiztonság" értelmével, amivel a Java a leghatékonyabban védi a kódodat. A Java a típusbiztonság révén garantálja, hogy nem rossz objektumot kérsz meg, hogy valami olyasmit csináljon, amit *valójában* egy másik objektumtípustól akartál kérni. Például nem kérhetsz meg egy *Ferrarit* (amiről azt hiszed, hogy egy *Kenyérpirító*), hogy *pirítson kenyeret.* Az igazság az, hogy még akkor *sem* kell az ilyen kínos helyzetek miatt aggódnod, ha mindenhez Object hivatkozásokat használsz, mert amikor egy objektumra Object hivatkozástípussal hivatkozol, a Java azt *hiszi*, hogy egy Object típusú objektumpéldányra mutatsz, ami azt jelenti, hogy csak azokat a tagfüggvényeket hívhatod meg az említett objektumra, amelyeket az Object osztály vezetett be.
Ha tehát ezt mondod...

```
Object o = new Ferrari();
o.goFast(); // Nem érvényes!
```

...akkor a fordítóprogramon sem jutsz túl.
Mivel a Java erősen típusos nyelv, a fordító mindig ellenőrzi, hogy egy tagfüggvényt olyan objektumra hívsz-e meg, amely tényleg képes *válaszolni.* Más szavakkal, egy tagfüggvényt *csak akkor* hívhatsz meg egy objektumhivatkozásra, ha a hivatkozott osztálytípus *rendelkezik* az említett tagfüggvénnyel. Ezzel később részletekbe menően foglalkozunk, úgyhogy ne aggódj, ha még nem látsz mindent kristálytisztán.

Az Object típusú többalakú hivatkozások használatának megvan az ára...

Mielőtt rohamtempóban nekiállnál az Object-et használni, mint ultrarugalmas argumentum- és visszatérési típust, nem árt, ha figyelembe veszel egy apróságot az Object típus hivatkozásként való használatával kapcsolatban. Ne feledd, hogy nem az Object típusú objektumpéldányok létrehozásáról beszélünk, hanem arról, amikor valamilyen más típusba tartozó objektumot hozol létre, de Object típusú hivatkozással.
Ha beleteszel egy objektumot egy ArrayList**<Dog>**-ba, akkor az Dog-ként megy be, és Dog-ként is jön ki:

```
ArrayList<Dog> myDogArrayList = new ArrayList<Dog>();
Dog aDog = new Dog();
myDogArrayList.add(aDog);
Dog d = myDogArrayList.get(0);
```

Egy Dog objektumok tárolására való ArrayList létrehozása

Egy Dog létrehozása

A Dog hozzáadása a listához

A listában található Dog hozzárendelése egy új, Dog típusú hivatkozó változóhoz (gondolj rá úgy, mintha a get() tagfüggvény a Dog visszatérési típust vezetné be, mert ArrayList<Dog>-ot használtál)

De mi történik, ha ezt így vezeted be: ArrayList**<Object>**? Ha olyan tömblistát szeretnél létrehozni, amely *tényleg* bármilyen fajta objektumot elfogad, így vezetheted be:

```
ArrayList<Object> myDogArrayList = new ArrayList<Object>();
Dog aDog = new Dog();
myDogArrayList.add(aDog);
```

Tetszőleges Object objektumok tárolására alkalmas ArrayList létrehozása

Egy Dog létrehozása

A Dog hozzáadása a listához

(Ez a két lépés megegyezik)

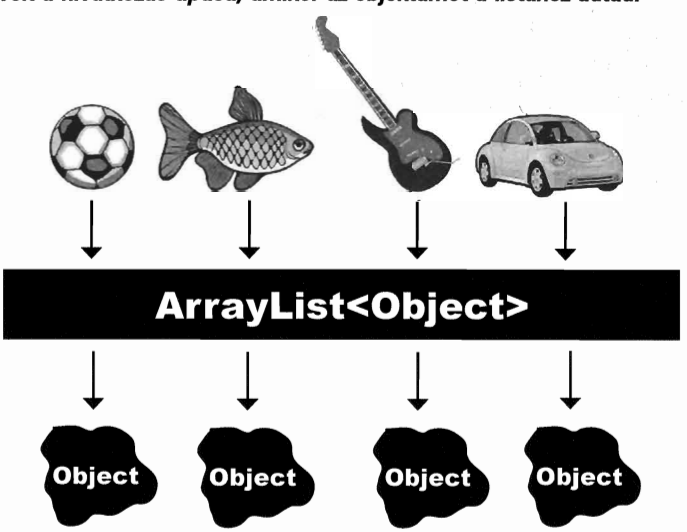
Mi történik, ha megpróbálod kiolvasni a Dog objektumot, és hozzárendelni egy Dog hivatkozáshoz?

```
Dog d = myDogArrayList.get(0);
```

NEM!!! Nem fordítható le!!!
Ha ArrayList<Object>-et használsz, a get() tagfüggvény Object típusú eredményt ad vissza. A fordító csak azt tudja, hogy az objektum örököl az Object-től (ott van valahol az öröklési fájában), de azt nem, hogy egy Dog-ról van szó!!!

Egy ArrayList<Object>-ből minden Object típusú hivatkozásként jön ki, függetlenül attól, hogy az objektum ténylegesen micsoda, vagy hogy mi volt a hivatkozás típusa, amikor az objektumot a listához adtad.

Az objektumok így mennek BE: **FociLabda**, **Hal**, **Gitár**, **Autó**.



Viszont úgy jönnek KI, mintha **Object** típusúak lennének.

Az objektumok úgy jönnek ki egy ArrayList<Object>-ből, mintha Object osztályú általános objektumpéldányok lennének. A fordító nem tételezhet fel semmilyen más típust a kijövő objektumokról, csak az Object-et.

Amikor egy Kutya nem úgy viselkedik, mint egy Kutya



Az a gond azzal, ha többalakú módon mindent Object-ként kezelünk, hogy az objektumok *látszólag* (de nem véglegesen) elveszítik a valódi énjüket. Úgy tűnik, a *Kutya elveszíti a kutyaságát.* Lássuk, mi történik, ha egy Dog-ot adsz át egy olyan tagfüggvénynek, amely egy ugyanarra a Dog objektumra mutató hivatkozást ad vissza, de a visszatérési típust nem Dog-ként, hanem Object-ként határozza meg:

☹ **ROSSZ**

```
public void go() {
    Dog aDog = new Dog();
    Dog sameDog = getObject(aDog);
}

public Object getObject(Object o) {
    return o;
}
```

Ez a sor nem fog működni! Bár a tagfüggvény ugyanarra a Dog-ra mutató hivatkozást adott vissza, mint amelyikre az argumentum hivatkozott, az Object visszatérési típus azt jelenti, hogy a fordító semmi máshoz nem engedi hozzárendelni a visszakapott hivatkozást, csak egy Object-hez.

Ugyanarra a Dog-ra mutató hivatkozást adunk vissza, de Object visszatérési típusúként. Ez a rész teljesen szabályszerű. Megjegyzés: ez hasonlít ahhoz, ahogy a get() tagfüggvény működik, amikor ArrayList<Dog> helyett ArrayList<Object>-et használunk.

```
File Edit Window Help Remember
DogPolyTest.java:10: incompatible types
found    : java.lang.Object
required: Dog
      Dog sameDog = takeObjects(aDog);
1 error                          ^
```

A fordító nem tudja, hogy a tagfüggvény által visszaadott dolog valójában egy Dog, ezért nem engedi hozzárendelni egy Dog hivatkozáshoz. (A következő oldalon azt is megtudod, miért.)

☺ **JÓ**

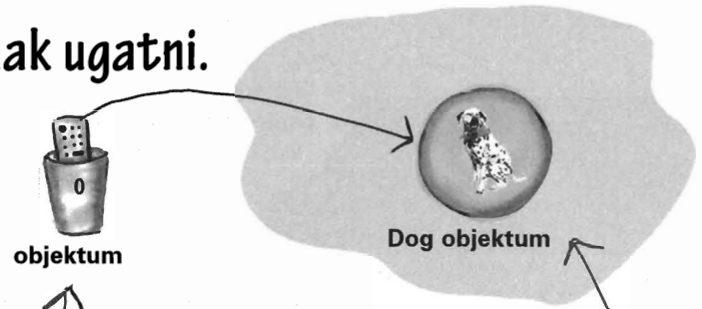
```
public void go() {
    Dog aDog = new Dog();
    Object sameDog = getObject(aDog);
}

public Object getObject(Object o) {
    return o;
}
```

Ez működik (bár nem valami hasznos, ahogy mindjárt látni fogod), mert egy Object típusú hivatkozáshoz BÁRMIT hozzárendelhetsz, hiszen minden osztály teljesíti az IS-A („egyfajta") tesztet az Object típusra. A Javában minden objektum az Object típus példánya, mert a Java minden osztálya végső soron az Object-től örököl.

Az Objektumok nem tudnak ugatni.

Most már tudod, hogy ha egy objektumra egy Object típusúként bevezetett változó hivatkozik, akkor az nem rendelhető hozzá olyan változóhoz, amelyet az objektum tényleges típusával vezettek be. Azt is tudod, hogy ez akkor fordulhat elő, ha egy visszatérési típust vagy argumentumot Object típusúként vezetsz be, például amikor az objektumot egy Object típusú tömblistába teszel az ArrayList<Object> utasítással. De milyen következményekkel jár mindez? Gondot okoz, ha Object típusú hivatkozó változót kell használnod, hogy egy Dog objektumra hivatkozhass? Próbáld meghívni a Dog tagfüggvényeit a fordító által Object-nek vélt Dog-ra:



Amikor egy objektumhivatkozást kapsz egy ArrayList<Object>-től (vagy bármely tagfüggvénytől, amely visszatérési típusként az Object-et adja meg), az Object típusú többalakú hivatkozásként érkezik. Esetünkben tehát egy Object típusú hivatkozásod lesz egy Dog-példányra.

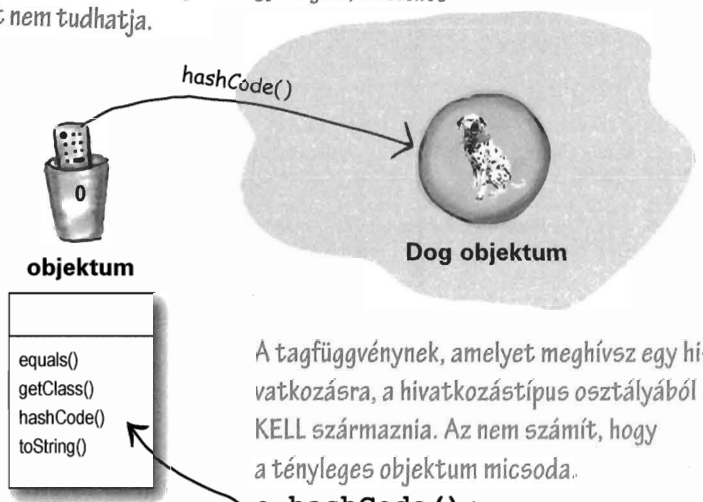
```
Object o = al.get(index);
int i = o.hashCode();
o.bark();
```

Ez rendben van. Az Object osztálynak van hashCode() tagfüggvénye, így ezt a tagfüggvényt BÁRMELY objektumra meghívhatod a Javában.

Nem fordítható le! Ezt nem lehet!!! Az Object osztálynak fogalma sincs róla, hogy mit jelent a bark(). Még ha TE tudod is, hogy a megadott sorszámú helyen valójában egy Dog áll, a fordító ezt nem tudhatja.

A fordító a *hivatkozás*, és nem a tényleges *objektum* típusa alapján dönti el, hogy meghívhatsz-e egy adott tagfüggvényt.

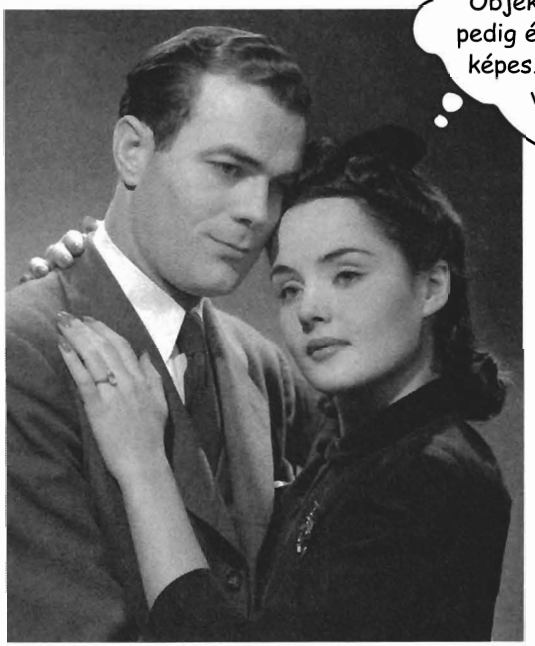
Még ha *tudod* is, hogy az objektum rendelkezik a szükséges képességgel („...de ez ***tényleg*** egy Kutya, becsszóra..."), a fordító csak egy általános Objektumot lát benne. A fordítótól lehetne egy Gomb vagy egy Mikrosütő is. Vagy bármi más, aminek gőze sincs arról, hogyan kell ugatni.
A fordító a *hivatkozás* típusának osztályát – és nem az *objektum* típusát – vizsgálja, hogy megállapítsa, hogy meghívhatsz-e egy tagfüggvényt az adott hivatkozásra.



A tagfüggvénynek, amelyet meghívsz egy hivatkozásra, a hivatkozástípus osztályából KELL származnia. Az nem számít, hogy a tényleges objektum micsoda.

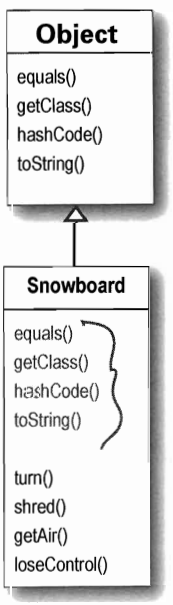
```
o.hashCode();
```

Az „o" hivatkozást Object típusúként vezettük be, ezért csak olyan tagfüggvényeket hívhatsz meg rá, amelyek megtalálhatók az Object osztályban.

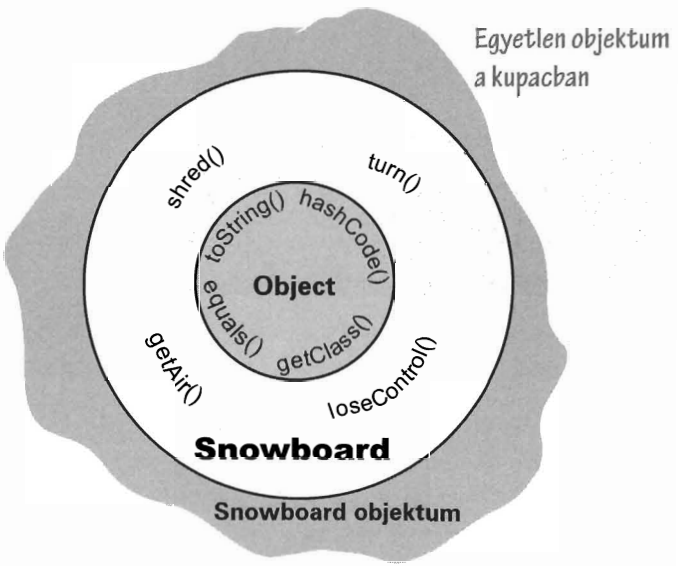


Lépj kapcsolatba a belső Objektumoddal!

Egy objektum *mindent* tartalmaz, amit az ősosztályaitól örököl. Ez azt jelenti, hogy *minden* objektum – a tényleges osztálytípusától függetlenül – *egyszersmind* az Object osztály példánya is. Vagyis a Javában minden objektum Object-ként is kezelhető, nem csak mint Dog, Button vagy Snowboard. Amikor kiadod azt az utasítást, hogy **`new Snowboard()`**, egyetlen objektumot kapsz a kupacban – egy Snowboard objektumot –, de az egy olyan belső magot vesz körbe, ami az objektum Object részét jelképezi.



A Snowboard örökli az Object alaposztály tagfüggvényeit, és hozzájuk ad még négyet.



Itt csak EGY objektum van a kupacban: egy Snowboard objektum. Ugyanakkor egyaránt tartalmazza önmagának a Snowboard osztályba, illetve az Object osztályba tartozó részét.

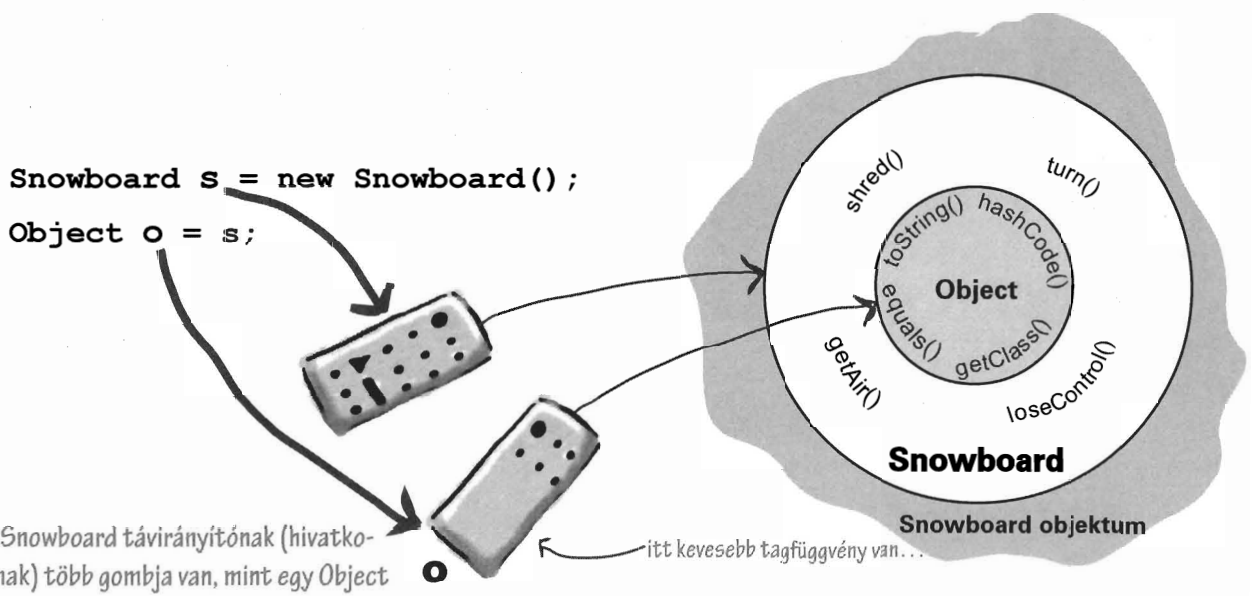
A „többalakúság" „több alakot" jelent

Egy Snowboard objektumot kezelhetsz Snowboard-ként, de Object-ként is.

Ha egy hivatkozás olyan, mint egy távirányító, akkor a távirányítóra egyre több gomb kerül, ahogy lefelé haladsz az öröklési fában. Egy Object típusú távirányítónak (hivatkozásnak) csak néhány gombja van – azok a gombok, amelyek az Object osztály elérhetővé tett tagfüggvényeihez tartoznak. Egy Snowboard típusú távirányító azonban amellett, hogy az Object osztály valamennyi gombját tartalmazza, a Snowboard típus új tagfüggvényeihez tartozó új gombokkal is kiegészíti azokat. Minél konkrétabb az osztály, annál több a gombja. Persze ez nem mindig igaz: előfordulhat, hogy egy alosztály egyetlen új tagfüggvényt sem tartalmaz, csupán felülbírálja az alaposztálya tagfüggvényeit. A lényeg az, hogy még ha az *objektum* Snowboard típusú is, egy a Snowboard objektumra mutató Object *hivatkozás* akkor sem láthatja a Snowboard sajátos tagfüggvényeit.

Ha egy objektumot egy ArrayList<Object>-be raksz, akkor csak Object típusúként kezelheted, függetlenül attól, hogy milyen típusú volt, amikor a listába tetted.

Ha egy hivatkozást kapsz vissza egy ArrayList<Object>-ből, annak a típusa mindig *Object* lesz, vagyis egy *Object* távirányítót kapsz.



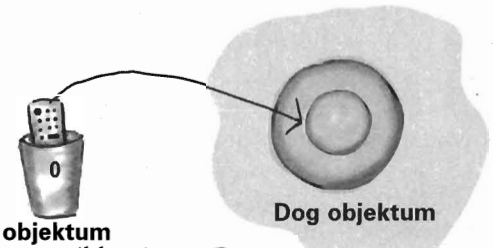
Egy Snowboard távirányítónak (hivatkozásnak) több gombja van, mint egy Object távirányítónak. A Snowboard távirányító a Snowboard objektum teljes „snowboardságát" láthatja, és a Snowboard valamennyi tagfüggvényéhez hozzáférhet, beleértve mind az Object-től örökölt tagfüggvényeket, mind a Snowboard osztály saját tagfüggvényeit.

Egy Object hivatkozás a Snowboard objektumnak csak az Object részeit láthatja, és csak az Object osztály tagfüggvényeihez férhet hozzá. Kevesebb gombja van, mint a Snowboard távirányítónak.



Alakítsd az úgynevezett „Objektumot" (amiről persze tudjuk, hogy valójában egy Kutya) Dog típusúvá, hogy valódi énjének megfelelően kezelhesd: Kutyaként.

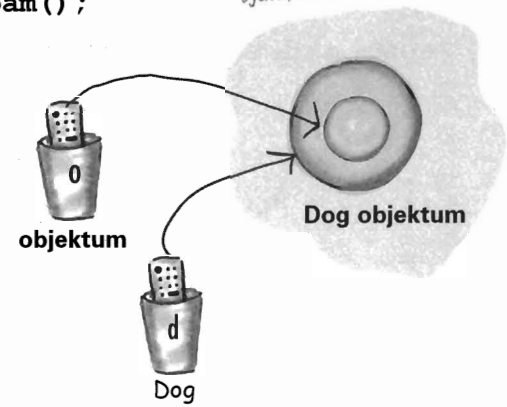
Objektumhivatkozás visszaalakítása a *valódi* típusára.



Valójában továbbra is egy Dog objektumról van szó, de ha a Dog sajátos tagfüggvényeit akarod meghívni rá, akkor egy Dog típusúként bevezetett *hivatkozásra* van szükséged. Ha *biztos* vagy benne*, hogy az objektum valójában Dog, létrehozhatsz egy rá mutató új Dog hivatkozást az Object hivatkozás lemásolásával és a másolatnak egy Dog típusú hivatkozó változóba kényszerítésével, vagyis a (Dog) típusátalakítás – cast – használatával. A *Dog* tagfüggvényeit ennek az új *Dog* hivatkozásnak a segítségével hívhatod meg:

```
Object o = al.get(index);
Dog d = (Dog) o;
d.roam();
```

az Objektumot visszaalakítjuk azzá a Kutyává, akiről tudjuk, hogy ott rejtőzik



*Ha nem vagy benne *biztos*, hogy Dog-ról van szó, az **instanceof** („példánya") művelettel ellenőrizd, mert ha tévedsz, amikor végrehajtod a típusátalakítást, futáskor ClassCastException kivételt kapsz, és a program lefagy.

```
if (o instanceof Dog) {
   Dog d = (Dog) o;
}
```

Most már tudod, mennyire ügyel a Java a hivatkozó változó osztályában levő tagfüggvényekre.

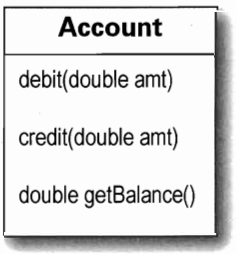
Egy tagfüggvényt *kizárólag* akkor hívhatsz meg egy objektumra, ha a hivatkozó változó osztálya rendelkezik az adott tagfüggvénnyel.

Az osztályod nyilvános tagfüggvényeire gondolj úgy, mint a szerződésedre, más szóval a külvilágnak tett ígéretedre arról, hogy mire vagy képes.



Amikor egy osztályt írsz, szinte mindig *felfedsz* néhányat a tagfüggvényei közül az osztályon kívül létező kódrészek számára. Egy tagfüggvényt *felfedni* annyit tesz, mint *elérhetővé tenni*, általában úgy, hogy nyilvánosként (public) jelölöd meg.

Tegyük fel, hogy egy kis méretű üzleti könyvelőprogram számára írsz kódot, mondjuk egy egyéni alkalmazást Simon szörfboltjához. Szorgos újrahasznosítóként találsz egy Account (Számla) nevű osztályt, amely – úgy tűnik – tökéletesen megfelel az igényeidnek, legalábbis a leírása szerint. Minden Account-példány egy-egy vásárlónak a boltban nyitott számláját jelképezi. Éppen meg akarod hívni a *credit()* és *debit()* tagfüggvényeket egy számlaobjektumra, amikor rájössz, hogy szükséged lesz a számlaegyenlegre. Semmi gond: van egy *getBalance()* – kiolvasEgyenleg – nevű tagfüggvény, az biztos jó lesz. Csakhogy amikor meghívod a *getBalance()* tagfüggvényt, a program futáskor bekrepál. Felejtsd el a dokumentációt, az osztályban nincs ilyen tagfüggvény. Gáz.



Ez viszont veled nem eshet meg, mert minden alkalommal, amikor a pont műveletet egy hivatkozásra alkalmazod (a.csinalValamit()), a fordítóprogram megvizsgálja a *hivatkozás* típusát (vagyis azt, hogy milyen típusúként vezették be az „a”-t), és ellenőrzi, hogy az osztály biztosan rendelkezik-e az adott tagfüggvénnyel, a tagfüggvény valóban elfogadja-e azt az argumentumot, amit átadsz neki, illetve olyan típusú értéket ad-e vissza, amilyet vársz.

Csak ne felejtsd el, hogy a fordító a *hivatkozó* változó, és nem a hivatkozás másik végén álló tényleges *objektum* osztályát vizsgálja.

Mi van, ha módosítanod kell a szerződést?

Tegyük fel, hogy Kutya vagy. A Dog osztályod nem az *egyetlen* szerződés, amely meghatározza a mibenlétedet. Ne feledd: minden ősosztályodtól örökölsz elérhető (ez általában *nyilvánost* jelent) tagfüggvényeket.
A Dog osztály valóban szerződésnek minősül, de nem a *teljes* szerződésnek.

A szerződésnek része minden olyan dolog is, ami a *Canine* osztályban található.
A szerződésnek része minden olyan dolog is, ami az *Animal* osztályban található.
A szerződésnek része minden olyan dolog is, ami az *Object* osztályban található.

Az IS-A teszt szerint ezek *mindegyike* vagy: Kutyaféle (Canine), Állat (Animal) és Objektum (Object).
De mi történik, ha az, aki az osztályodat tervezte, egy állatszimulációs programban akar felhasználni (téged, mint Dog osztályt), hogy bemutatót tarthasson az Animal objektumok helyes tartásáról?
Nos, ezzel semmi gond, ilyen célra valószínűleg újrahasznosítható vagy.
És ha később egy kisállat-kereskedési programhoz akar felhasználni? *Nincsenek házikedvenc-viselkedéseid.* Egy Pet (Házikedvenc) osztályban olyan tagfüggvényekre van szükség, mint a barátságos viselkedést biztosító *beFriendly()* vagy a játékra való *play()*.
Oké, tegyük fel, hogy te programozod a Dog osztályt. Nem gond, ugye? Egyszerűen hozzáadsz néhány további tagfüggvényt a Dog osztályhoz. Tagfüggvények *hozzáadásával* nem teszed tönkre senkinek a kódját, hiszen nem nyúlsz a *meglevő* tagfüggvényekhez, amelyeket más kódok a Dog objektumokra meghívhatnak.
Látod bármilyen hátrányát ennek a megoldásnak (mármint annak, hogy Pet-tagfüggvényeket adsz a Dog osztályhoz)?



Agytorna

Gondold végig, hogy **TE** mit tennél, ha **TE** lennél a Dog osztály programozója, és úgy kellene módosítanod a Dog osztályt, hogy Pet-féle dolgokat is tudjon csinálni. Azt tudod, hogy ha egyszerűen új Pet-viselkedéseket (tagfüggvényeket) adsz a Dog osztályhoz, az működik, és nem teszi tönkre senkinek a kódját.

Csakhogy ez egy házikedvenc-program, amelyben nem csak kutyák vannak. Mi történik, ha valaki egy olyan programhoz akarja felhasználni a Dog osztályodat, amelyben *vad* Kutyák vannak? Gondold végig, milyen lehetőségeid vannak: azzal ne foglalkozz, hogy a Java mit hogyan kezel, csak arra összpontosíts, hogy te hogyan *szeretnéd* megoldani az Animal-osztályok módosítását, hogy Pet-viselkedések is legyenek bennük.

Állj meg egy percre, és gondolkodj, **mielőtt a következő oldalra pillantanál**, ahol *minden* titokról fellibbentjük a fátylat.

(Ne less, különben az egész gyakorlatnak semmi értelme, és megfosztod magad a Nagy Lehetőségtől, hogy az agyaddal elégess néhány kalóriát.)

Lássunk néhány tervezési lehetőséget a meglevő osztályaink újrahasznosítására egy házikedvenc-programban!

A következő néhány oldalon végigveszünk néhány megoldási lehetőséget. Azzal egyelőre nem foglalkozunk, hogy a Java valóban képes-e *megcsinálni*, amit kitaláltunk, de ha már látjuk az egyes megoldások előnyeit és hátrányait, erre is sort kerítünk.

① Első lehetőség

Az egyszerű megoldást választod, és a házikedvenc-tagfüggvényeket az Animal osztályban helyezed el.

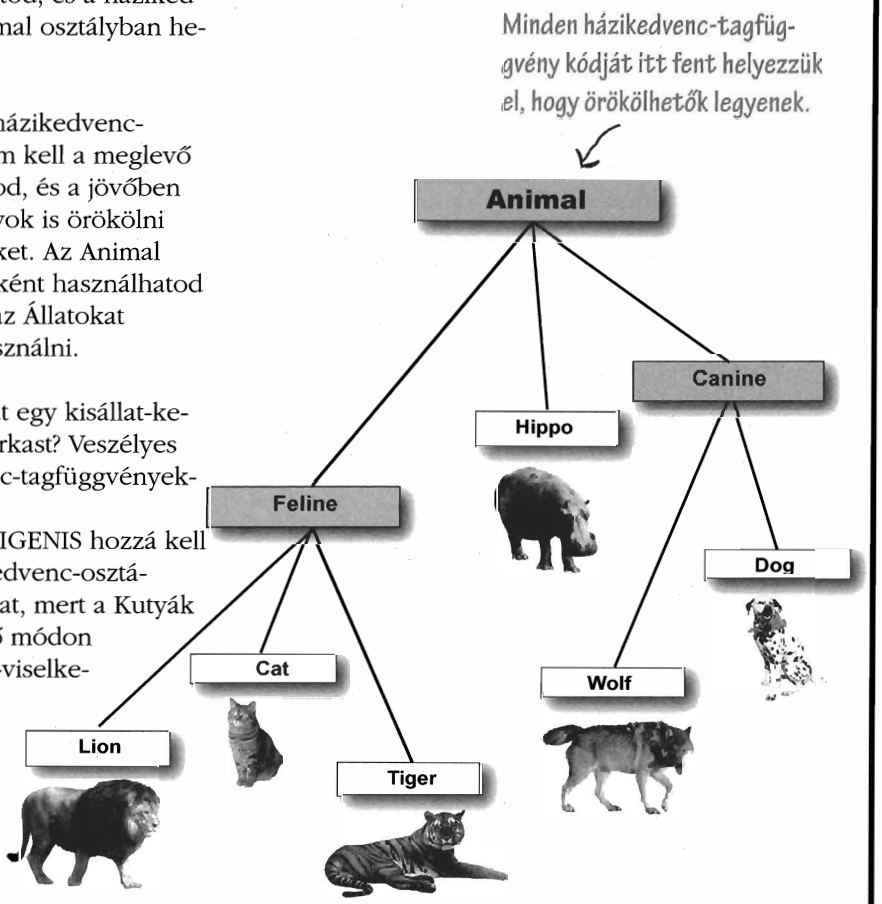
Előnyök:

Minden Állat rögtön örökli a házikedvenc-viselkedéseket. Egyáltalán nem kell a meglevő Animal-alosztályokhoz nyúlnod, és a jövőben létrehozandó Animal-alosztályok is örökölni fogják ezeket a tagfüggvényeket. Az Animal osztályt tehát többalakú típusként használhatod bármely programban, amely az Állatokat házikedvencként akarja felhasználni.

Hátrányok:

Mikor is láttál utoljára vízilovat egy kisállat-kereskedésben? És oroszlánt? Farkast? Veszélyes lehet vadállatokat házikedvenc-tagfüggvényekkel ellátni.

Ezenkívül szinte biztos, hogy IGENIS hozzá kell majd nyúlnod az olyan házikedvenc-osztályokhoz, mint a Dog vagy a Cat, mert a Kutyák és a Macskák NAGYON eltérő módon valósítják meg a házikedvenc-viselkedéseket (a mi otthonunkban legalábbis így van).



② Második lehetőség

Az első megoldásból kiindulva a házikedvenc-tagfüggvényeket az Animal osztályban helyezed el, de a tagfüggvényeket elvonttá teszed, arra kényszerítve az Animal alosztályait, hogy felülbírálják azokat.

ELŐNYÖK:

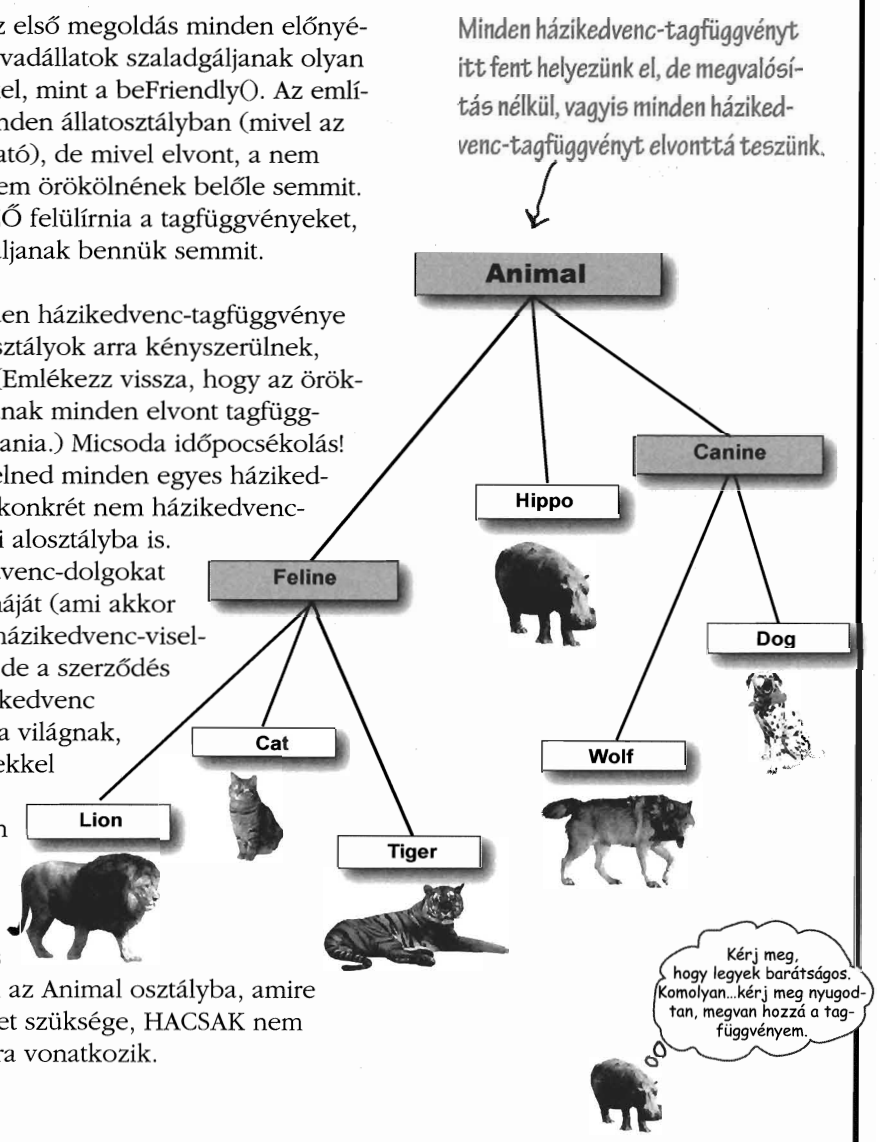
Ez a megoldás rendelkezik az első megoldás minden előnyével, de megkímél attól, hogy vadállatok szaladgáljanak olyan házikedvenc-tagfüggvényekkel, mint a beFriendly(). Az említett tagfüggvény ott lenne minden állatosztályban (mivel az Animal osztályban megtalálható), de mivel elvont, a nem házikedvenc állatosztályok nem örökölnének belőle semmit. Minden osztálynak KÖTELEZŐ felülírnia a tagfüggvényeket, de megtehetik, hogy ne csináljanak bennük semmit.

HÁTRÁNYOK:

Mivel az Animal osztály minden házikedvenc-tagfüggvénye elvont, a konkrét Animal-alosztályok arra kényszerülnek, hogy mindet megvalósítsák. (Emlékezz vissza, hogy az öröklési fa első konkrét alosztályának minden elvont tagfüggvényt KÖTELEZŐ megvalósítania.) Micsoda időpocsékolás! Le kell ülnöd, és be kell gépelned minden egyes házikedvenc-tagfüggvényt az összes konkrét nem házikedvenc-osztályba, és minden jövőbeli alosztályba is.

Ez megoldja ugyan a házikedvenc-dolgokat CSINÁLÓ vadállatok problémáját (ami akkor merülne fel, ha örökölnék a házikedvenc-viselkedést az Animal osztálytól), de a szerződés rossz, mert minden *nem* házikedvenc osztálynak be kell jelentenie a világnak, hogy maga is rendelkezik ezekkel a tagfüggvényekkel, annak ellenére, hogy azok valójában nem CSINÁLNAK semmit, amikor meghívják őket.

Ez a megközelítés egyáltalán nem tűnik jónak. Nem helyes minden olyasmit belezsúfolni az Animal osztályba, amire egynél több állattípusnak lehet szüksége, HACSAK nem az Animal ÖSSZES alosztályára vonatkozik.



③ Harmadik lehetőség

A házikedvenc-tagfüggvényeket CSAK azokban az osztályokban helyezed el, ahol tényleg szükség van rájuk.

Előnyök:

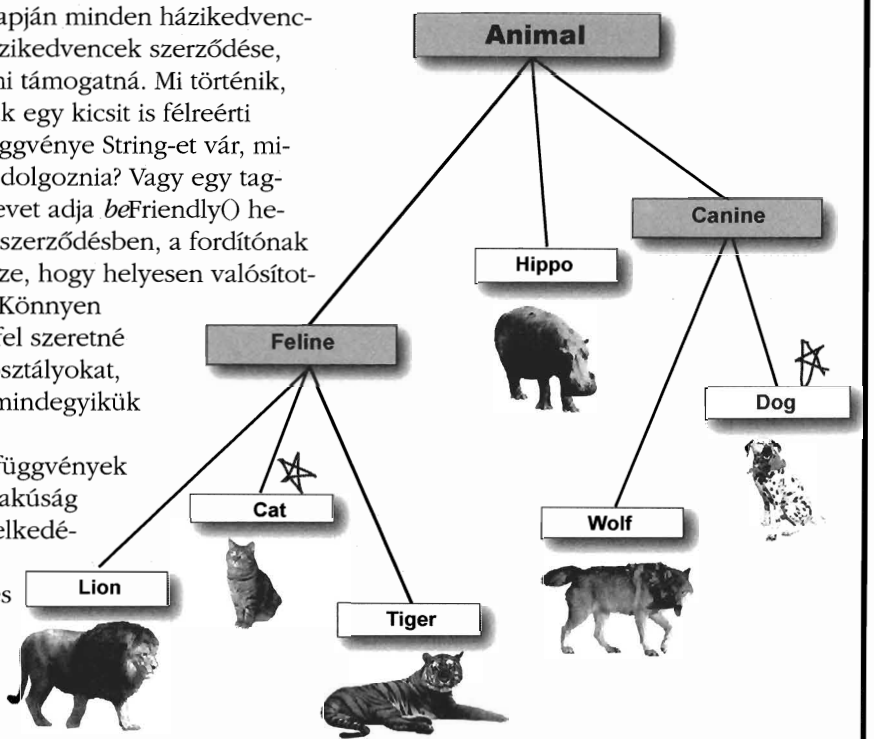
Így nem kell többé aggódni amiatt, hogy egy víziló köszönt az ajtóban, vagy nyalogatja az arcod. A tagfüggvények oda és CSAK oda kerülnek, ahová valók. A Kutyák és a Macskák megvalósíthatják a tagfüggvényeket, de senki másnak nem kell tudnia róluk.

Hátrányok:

Ezzel a megközelítéssel két Nagy Probléma van. Először is, ki kell dolgoznod egy protokollt, amelyről a házikedvenc állatosztályok jelenlegi és jövőbeli programozóinak TUDNIUK kell. Protokoll alatt azokat a tagfüggvényeket értjük, amelyekkel a döntés alapján minden házikedvencnek rendelkeznie kell. Ez a házikedvencek szerződése, ami mögött semmi nem áll, ami támogatná. Mi történik, ha valamelyik programozó csak egy kicsit is félreérti a protokollt, például egy tagfüggvénye String-et vár, miközben int értékekkel kellene dolgoznia? Vagy egy tagfüggvénynek a *do*Friendly() nevet adja *be*Friendly() helyett? Mivel ez nem szerepel a szerződésben, a fordítónak nincs módja rá, hogy ellenőrizze, hogy helyesen valósítottad-e meg a tagfüggvényeket. Könnyen előfordulhat, hogy valaki, aki fel szeretné használni a házikedvenc állatosztályokat, egyszercsak rájön, hogy nem mindegyikük működik tökéletesen.

Másodszor, a házikedvenc-tagfüggvények esetében nem élhetsz a többalakúság előnyeivel. A házikedvenc-viselkedéseket igénylő osztályok mindegyikének tudnia kell az összes osztályról! Más szavakkal, az Animal-t ennél a megoldásnál nem alkalmazhatod többalakú típusként, mert a fordító nem engedi meg, hogy egy házikedvenc-tagfüggvényt egy Animal hivatkozásra hívj meg (még akkor sem, ha valójában egy Dog objektumról van szó), mert az Animal osztály nem rendelkezik a tagfüggvényel.

A házikedvenc-tagfüggvényeket az Animal helyett CSAK azokban az állatosztályokban helyezzük el, amelyek lehetnek házikedvencek.

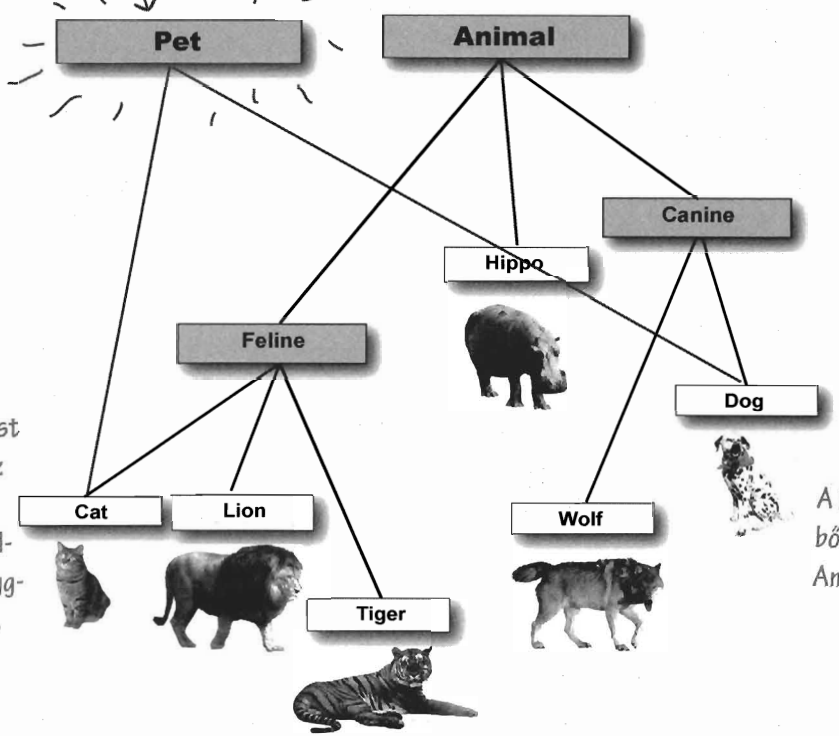


Amire tehát VALÓJÁBAN szükséged van:

- Valahogy el kell érned, hogy **csak** a házikedvenc-osztályok mutassanak házikedvenc-viselkedést.
- Valahogy garantálnod kell, hogy minden házikedvenc-osztály ugyanazokat a tagfüggvényeket határozza meg (ugyanazon a néven, ugyanazokkal az argumentumokkal és visszatérési típusokkal, ne hiányozzon egyetlen tagfüggvény sem, és így tovább), és ne kelljen imádkoznod, hogy minden programozó helyesen értelmezze a protokollt.
- Valahogy ki kell használnod a többalakúság előnyeit, hogy minden házikedvenc-osztály házikedvenc-tagfüggvényeit meghívhasd anélkül, hogy argumentumokat, visszatérési típusokat és tömböket kellene használnod minden egyes házikedvenc-osztályhoz.

Úgy fest, KÉT alaposztályra lesz szükség a fa tetején.

Létrehozunk egy új elvont alaposztályt Pet néven, és ebbe tesszük az összes házikedvenc-tagfüggvényt.



A Cat osztály most egyaránt bővíti az Animal ÉS a Pet osztályt, így mindkettőnek a tagfüggvényeit megkapja.

A Dog egyaránt bővíti a Pet és az Animal osztályt.

A vadállatok semmilyen házikedvenc-dolgot nem örökölnek.

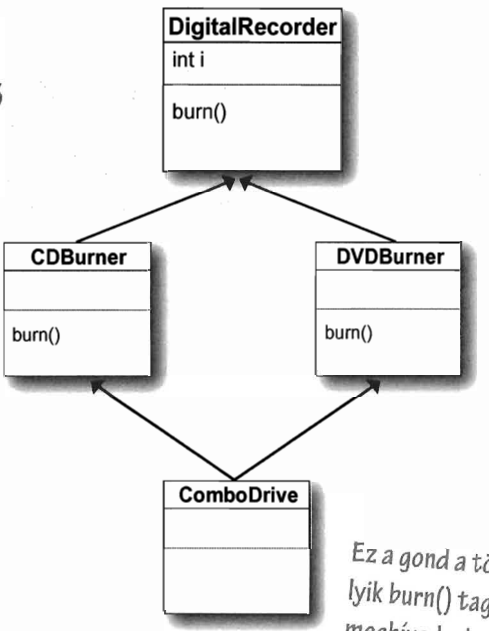
A „két alaposztály" módszerrel csak egy baj van...

Ezt többszörös öröklésnek hívják, és néha Nagyon Rossz Dolog.

...lenne, ha lehetséges lenne a Javában.
Csakhogy tilos, mert a többszörös öröklésnek van egy rákfenéje:
A Halál Végzetes Gyémántja (The Deadly Diamond of Death, DDD).

A Halál Végzetes Gyémántja

A CDBurner és a DVDBurner egyaránt örököse a DigitalRecorder-nek, és mindkettő felülírja a burn() tagfüggvényt, illetve örökli az „i" példányváltozót.



Tegyük fel, hogy az „i" példányváltozót a CDBurner és a DVDBurner osztály is felhasználja, de különböző értékkel. Mi történik, ha a ComboDrive osztálynak az „i" mindkét értékére szüksége van?

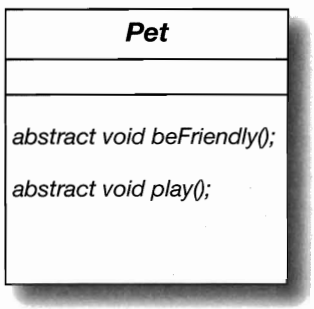
Ez a gond a többszörös örökléssel. Melyik burn() tagfüggvény fut le, amikor meghívod a burn()-t a ComboDrive-ra?

Egy olyan nyelvben, amely megengedi A Halál Végzetes Gyémántját, csúnya bonyodalmak adódhatnak, mivel az esetleges kétértelműséget külön szabályokkal kell eloszlatnod. A több szabály pedig azt jelenti, hogy többet kell dolgoznod: egyrészt azért, hogy *kiderítsd* ezeket a szabályokat, másrészt pedig azért, hogy felkészülj a „különleges esetekre". A Javát viszont *egyszerűnek* szánták, ahol egységes szabályok vannak, amelyek nem borulnak fel bizonyos körülmények között. Ezért a Java (ellentétben a C++-szal) megkímél attól, hogy A Halál Végzetes Gyémántján kelljen töprengened. Így viszont visszajutsz az eredeti problémához: *hogyan kezeljük az „Állat vagy Házikedvenc" problémát?*

Felületek: a felmentő sereg

A Java kínál egy megoldást. Használj egy *felületet* (interface). Nem grafikus felhasználói felületre (GUI) gondolunk, és nem is a felület *szó* általános jelentésére („Ez a Button osztály API-jának nyilvános felülete"), hanem a Java **`interface`** *kulcsszavára.*

A Java-felületek olyan megoldást nyújtanak a többszörös öröklés problémájára, amely biztosítja a többszörös öröklésben foglalt többalakúság *előnyeit*, miközben megkímél A Halál Végzetes Gyémántja okozta fájdalomtól és szenvedéstől. Az, ahogy a felületek megkerülik A Halál Végzetes Gyémántját, meglepően egyszerű: ***minden tagfüggvényt elvonttá tesznek***! Az alosztályoknak tehát meg ***kell*** valósítaniuk a tagfüggvényeket (emlékezz vissza, hogy az elvont tagfüggvényeket az első konkrét alosztálynak *kötelező* megvalósítania), így a JVM nem zavarodik össze, hogy a két örökölt változat közül *melyiket* kell meghívnia.



A Java-felületek olyanok, mint egy 100%-ig tiszta elvont osztály.

Egy felületben minden tagfüggvény elvont, ezért minden olyan osztálynak, amely „egyfajta" (IS-A) Pet, KÖTELEZŐ megvalósítania (vagyis felülírnia) a Pet tagfüggvényeit.

Így fest egy felület MEGHATÁROZÁSA:

```
public interface Pet {...}
```

A „class" helyett az „interface" kulcsszót kell használnod.

És így fest egy felület MEGVALÓSÍTÁSA:

```
public class Dog extends Canine implements Pet {...}
```

Itt az „implements" kulcsszót kell használnod, amit a felület neve követ. Vedd észre, hogy amikor megvalósítasz egy felületet, továbbra is bővíthetsz egy osztályt.

A Pet felület meghatározása és megvalósítása

A felületek tagfüggvényei automatikusan nyilvánosak és elvontak lesznek, ezért a „public" és „abstract" kulcsszavak beírása nem kötelező (sőt nem is minősül „jó stílusnak", ha beírod őket – mi itt csak azért tettünk így, hogy világosabb legyen, és persze azért, mert soha nem voltunk a divat rabjai...)

Itt „class" helyett azt mondod, hogy „interface".

```
public interface Pet {

    public abstract void beFriendly();

    public abstract void play();

}
```

Minden felületi tagfüggvény elvont, ezért KÖTELEZŐ pontosvesszőt tenni utánuk. Ne feledd, hogy nincs törzsük!

A Kutya egyfajta Állat (Dog IS-A Animal), és a Kutya egyfajta Házikedvenc (Dog IS-A Pet).

Azt mondod, hogy „implements", amit a felület neve követ.

```
public class Dog extends Canine implements Pet {

    public void beFriendly() {...}

    public void play() {..}


    public void roam() {...}

    public void eat() {...}

}
```

Azt MONDTAD, hogy Pet vagy, ezért meg KELL valósítanod a Pet tagfüggvényeit. Ez a szerződésed. Figyeld meg, hogy itt pontosvesszők helyett kapcsos zárójelek állnak.

Ezek szokványos felülíró tagfüggvények.

Nincsenek

hülye kérdések

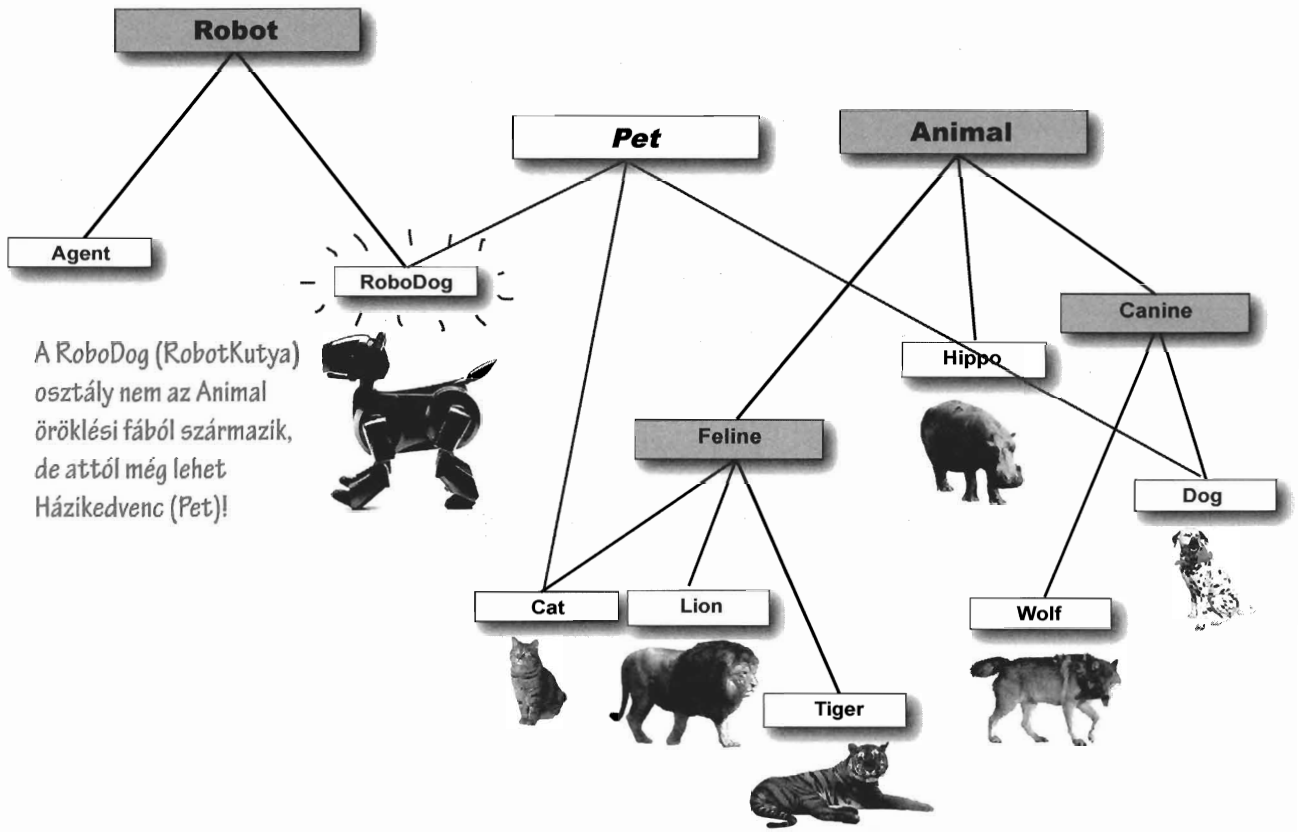
K: Várjunk csak egy percet! A felületek nem biztosítanak valódi többszörös öröklést, hiszen nem tehetünk beléjük semmilyen megvalósító kódot. Ha minden tagfüggvény elvont, akkor valójában milyen előnyöket nyújt egy felület?

V: Többalakúság, többalakúság, többalakúság. A felületeknél nincs rugalmasabb: ha argumentumként és visszatérési típusként felületeket használsz konkrét alosztályok (vagy akár elvont alaposztályok) helyett, bármit átadhatsz, ami megvalósítja az adott felületet. És gondolj csak bele – ha felületet használsz, az osztályoknak nem muszáj ugyanabból az öröklési fából származniuk. Egy osztály bővíthet egy másikat, és megvalósíthat egy felületet, miközben egy másik osztály ugyanazt a felületet valósítja meg, pedig teljesen más öröklési fában található. Az objektumokat tehát a szerepük szerint kezelheted, nem pedig annak az osztálynak a típusa szerint, amelyikből példányosították őket.

Valójában, ha úgy írod meg a kódod, hogy felületeket használjon, nincs is szükség semmilyen alaposztályra, amit bővíteni kellene. Egyszerűen bemutatod a felületet az osztályoknak, és azt mondod nekik: „Nem érdekel, milyen osztályöröklési szerkezetből származtok, csak valósítsátok meg ezt a felületet, és jóban leszünk."

Az, hogy egy felületbe nem tehetsz megvalósító kódot, a legtöbb esetben nem gond, mert a legtöbb felületi tagfüggvényt amúgy sem lenne értelme általánosan megvalósítani. Más szavakkal, a felületi tagfüggvények többségét felül kell bírálni, még akkor is, ha azokat nem *kényszerítettük* arra, hogy elvontak legyenek.

Ugyanazt a felületet *különböző* öröklési fából származó osztályok is megvalósíthatják.



Amikor egy *osztályt* használsz többalakú típusként (például Canine típusú argumentumokat váró tagfüggvényként vagy Animal típusú tömbként), az említett típusba csak olyan objektumokat szúrhatsz be, amelyek ugyanabból az öröklési fából származnak. Az sem mindegy, hogy az öröklési fa melyik részéből jönnek: az objektumoknak egy olyan osztályból kell származniuk, amely a többalakú típus alosztálya. Egy Canine típusú paraméter elfogad egy Wolf vagy egy Dog argumentumot, de egy Cat-et vagy Hippo-t nem.

Amikor azonban **felületet** használsz többalakú típusként (például házikedvencek tömbjeként), az objektumok az öröklési fa *bármely* részéből származhatnak.

Az egyetlen követelmény az, hogy olyan osztály legyen a forrásuk, amely *megvalósítja* az adott felületet. A Java API kulcsfontosságú jellemzője, hogy megengedi, hogy egy közös felületet különböző öröklési fák osztályai is megvalósíthassanak. Azt szeretnéd, hogy egy objektum képes legyen az állapotát fájlba menteni? Valósítsd meg a Serializable („sorosítható") felületet! Azt szeretnéd, hogy az objektumok külön végrehajtási szálon futtassák a tagfüggvényeiket? Valósítsd meg a Runnable („futtatható") felületet! Érted a lényeget. A Serializable és Runnable felületekről a későbbi fejezetekben többet is megtudhatsz; egyelőre elég annyit megjegyezned, hogy az öröklési fa *bármelyik* részéből származó osztályok megvalósíthatják ezeket a felületeket. Majdnem *minden* osztály lehet menthető vagy futtatható.

Ami még jobb: egy osztály *több* felületet is megvalósíthat!

A Kutya egyfajta Kutyaféle, egyfajta Állat és egyfajta Objektum – mindezt az öröklés teszi lehetővé. A Kutya azonban egyfajta Házikedvenc is lehet egy felület megvalósításával, és a Kutya más felületeket is megvalósíthat. Mondhatod például ezt:

```
public class Dog extends Animal implements
Pet, Saveable, Paintable { ... }
```



Honnan tudhatod, hogy osztályt, alosztályt, elvont osztályt vagy felületet célszerű létrehoznod?

- Ha az új osztály egyetlen más típusra sem teljesíti az IS-A tesztet, hozd létre úgy, hogy ne bővítsen semmit (az Object-en kívül).
- Alosztályt csak akkor hozz létre (vagyis egy osztályt csak akkor *bővíts*), ha egy osztály ***konkrétabb*** változatára van szükséged, új vagy felülírt viselkedésekkel.
- Elvont osztályt akkor használj, ha egy ***sablont*** akarsz meghatározni alosztályok egy csoportja számára, és legalább némi megvalósító kóddal rendelkezel, amelyet minden alosztály használhat. Akkor tegyél elvonttá egy osztályt, ha garantálni szeretnéd, hogy senki nem hozhat létre ilyen típusú objektumot.
- Felületet akkor használj, ha egy olyan ***szerepet*** szeretnél meghatározni, amelyet különféle osztályok tölthetnek be, az öröklési fában elfoglalt helyüktől függetlenül.

Tagfüggvény alaposztálybeli változatának meghívása

K: Mi van, ha létrehozok egy konkrét alosztályt, és felül kell írnom egy tagfüggvényt, de a tagfüggvény alaposztálybeli változatának viselkedésére van szükségem? Más szóval, mit tehetek, ha nem akarom *lecserélni* a tagfüggvényt egy felülíró változattal, csak *ki szeretném egészíteni* valamilyen konkrét kóddal?

V: Gondolj az „extends" („bővít") szó jelentésére. A jó objektumközpontú tervezés egyik pillére, hogy miként tervezhetsz olyan konkrét kódot, aminek az a *célja*, hogy felülbírálják. Más szavakkal, egy tagfüggvénybe olyan kódot írsz – mondjuk egy elvont osztályban –, ami elég általános ahhoz, hogy támogassa a konkrét megvalósításokat, de a konkrét kód nem képes az alosztály *minden* feladatát elvégezni, ezért az alosztály felülbírálja a tagfüggvényt, és *kibővíti* azt a szükséges kóddal.
A super kulcsszó segítségével az alosztályból hívhatod meg a felülírt tagfüggvény alaposztálybeli változatát.

a tagfüggvény alaposztálybeli változata fontos dolgokat csinál, amire az alosztályoknak szükségük lehet

```
abstract class Report {
    void runReport() {
        // beállítás
    }
    void printReport() {
        // általános nyomtatás
    }
}

class BuzzwordsReport extends Report {

    void runReport() {
        super.runReport();
        buzzwordCompliance();
        printReport();

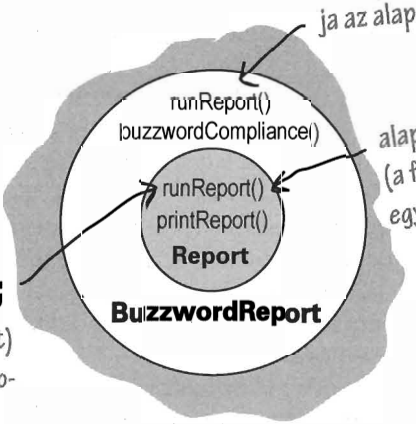
    }
    void buzzwordCompliance() {...}
}
```

meghívjuk az alaposztálybeli változatot, majd visszatérünk, és csinálunk valamit, ami az alosztályra jellemző

Ha a BuzzwordReport egyik alosztályában a tagfüggvény kódja ezt mondja:

```
super.runReport();
```

akkor a Report alaposztály runReport() tagfüggvénye fog lefutni.



super.runReport();

Egy alosztálybeli objektumra (BuzzwordReport) mutató hivatkozás mindig az alosztálybeli változatát hívja meg a felülírt tagfüggvénynek.
Ez a többalakúság. Az alosztály kódja azonban a super.runReport() utasítással meghívhatja az alaposztálybeli változatot.

A super kulcsszó valójában egy hivatkozás egy objektum alaposztálybeli részére. Ha egy alosztálybeli kód a super kulcsszót használja – mint a super.runReport()-ban – a tagfüggvény alaposztálybeli változata fog lefutni.

Újratöltve



- Ha nem akarod, hogy egy osztályból példányokat hozhassanak létre (vagyis nem szeretnéd, hogy bárki új objektumot hozzon létre az adott osztálytípusból), jelöld meg az osztályt az **abstract** kulcsszóval.
- Az elvont osztályoknak elvont és nem elvont tagfüggvényeik is lehetnek.
- Ha egy osztálynak akár csak *egy* elvont tagfüggvénye is van, az osztályt kötelező elvonttá tenni.
- Az elvont tagfüggvényeknek nincs törzsük, és a bevezetésüket pontosvesszővel kell lezárni (kapcsos zárójelek nélkül).
- Minden elvont tagfüggvényt meg kell valósítani az öröklési fa első konkrét alosztályában.
- A Javában minden osztály közvetlen vagy közvetett alosztálya az **Object** (java.lang.Object) osztálynak.
- A tagfüggvények Object típusú argumentumokkal, illetve visszatérési típusokkal is bevezethetők.
- Egy objektumra csak akkor hívhatsz meg egy tagfüggvényt, ha a tagfüggvény megtalálható az osztályban (vagy felületben) a *hivatkozó* változó típusaként, függetlenül a *tényleges* objektum típusától. Egy Object típusú hivatkozó változó tehát csak az Object osztályban meghatározott tagfüggvények meghívására használható, függetlenül attól, hogy mi a típusa annak az objektumnak, amelyre a hivatkozás mutat.
- Egy Object típusú hivatkozó változó nem rendelhető hozzá semmilyen más hivatkozástípushoz *típusátalakítás* (cast) nélkül. Típusátalakítással hozzárendelhetsz egy adott típusú hivatkozó változót egy annak altípusába tartozó hivatkozó változóhoz, de futásidőben a típusátalakítás meghiúsul, ha a kupacban levő objektum NEM összeegyeztethető típusú az átalakítás céltípusával. Például: `Dog d = (Dog) x.getObject(aDog);`
- Egy ArrayList<Object>-ből minden objektum Object típusúként távozik (ami azt jelenti, hogy csak Object típusú hivatkozó változóval hivatkozhatsz rájuk, hacsak nem alkalmazol *típusátalakítást*).
- A Halál Végzetes Gyémántja okozta problémák miatt a többszörös öröklés nem megengedett a Javában. Ez azt jelenti, hogy csak egy osztályt bővíthetsz (vagyis egy osztálynak csak egy közvetlen őse lehet).
- A felületek 100%-os tisztaságú elvont osztályok, amelyek *kizárólag* elvont tagfüggvényeket határoznak meg.
- Ha felületet szeretnél létrehozni, a **class** helyett az **interface** kulcsszót kell használnod.
- Egy felületet az implements kulcsszóval valósíthatsz meg. Például: `Dog implements Pet`
- Egy osztály több felületet is megvalósíthat.
- Egy felületet megvalósító osztálynak a felület minden tagfüggvényét *kötelező* megvalósítania, mivel ***minden felületi tagfüggvény automatikusan nyilvános és elvont.***
- Ha egy tagfüggvénynek az alaposztálybeli változatát szeretnéd meghívni egy olyan alosztályból, amely felülbírálja a tagfüggvényt, a **super** kulcsszót használhatod. Például: `super.runReport();`

K: Valami még mindig nem stimmel itt...Azt sehol nem magyaráztátok el, hogy miként lehetséges, hogy az ArrayList<Dog> olyan Dog hivatkozásokat ad vissza, amelyeken nem kell típusátalakítást végezni, miközben az ArrayList osztály Object-eket használ a tagfüggvényeiben, és nem Dog-okat (vagy DotCom-okat, vagy bármi mást). Van valamilyen különleges trükkje az ArrayList<Dog>-nak?

V: Igazad van, ez tényleg egy különleges trükk, hiszen az ArrayList<Dog> úgy ad vissza Dog-okat, hogy nem kell típusátalakítást végezned, pedig az ArrayList tagfüggvényei látszólag semmit sem tudnak a Dog-okról, sőt az Object-en kívül semmilyen típusról. A válasz röviden az, hogy a *fordítóprogram szúrja be helyetted a típusátalakítást.* Amikor azt mondod, hogy ArrayList<Dog>, nincs semmilyen különleges osztály, amelynek tagfüggvényei lennének a Dog objektumok fogadására és visszaadására, viszont a <Dog> jelzés a fordítónak, hogy azt szeretnéd, ha a fordító megengedné, hogy CSAK Dog objektumokat tegyél a listába, és megakadályozna abban, hogy bármilyen más típussal próbálkozz. Mivel pedig a fordító nem hagyja, hogy Dog-okon kívül bármi mást adj az ArrayList-hez, a fordító azt is tudja, hogy biztonságosan Dog hivatkozássá alakíthat bármit, ami az ArrayList-ből származik. Más szavakkal, az ArrayList<Dog> utasítás megkímél attól, hogy át kelljen alakítanod a visszakapott Dog típusát. Van azonban valami, ami még ennél is lényegesebb...Ne feledd, hogy előfordulhat, hogy egy típusátalakítás futásidőben meghiúsul – nem szeretnéd, hogy a hibákra inkább fordításkor derüljön fény, mint mondjuk akkor, amikor a felhasználó valamilyen fontos műveletet próbál végrehajtani? Erről a témáról még bőven van mit mondani – a részletekben el is merülünk majd a gyűjteményekről szóló fejezetben.



Gyakorlat

Itt az alkalom, hogy megmutasd a művészi képességeidet! A bal oldalon osztályok és felületek bevezetéseit láthatod. A feladatod az, hogy jobboldalt megrajzold a hozzájuk tartozó osztálydiagramokat. Az elsőt megcsináltuk helyetted. A „megvalósítja" (implements) jelölésére használj szaggatott, a „bővíti" (extends) jelölésére pedig megszakítatlan vonalakat.

Az adott kód:

1)
```
public interface Foo { }
public class Bar implements Foo { }
```

2)
```
public interface Vinn { }
public abstract class Vout implements Vinn { }
```

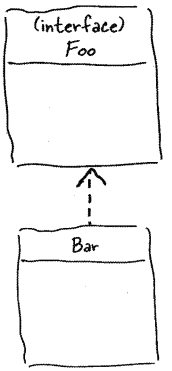
3)
```
public abstract class Muffie implements Whuffie { }
public class Fluffie extends Muffie { }
public interface Whuffie { }
```

4)
```
public class Zoop { }
public class Boop extends Zoop { }
public class Goop extends Boop { }
```

5)
```
public class Gamma extends Delta implements Epsilon { }
public interface Epsilon { }
public interface Beta { }
public class Alpha extends Gamma implements Beta { }
public class Delta { }
```

Legyen képed hozzá!

1)



2)

3)

4)

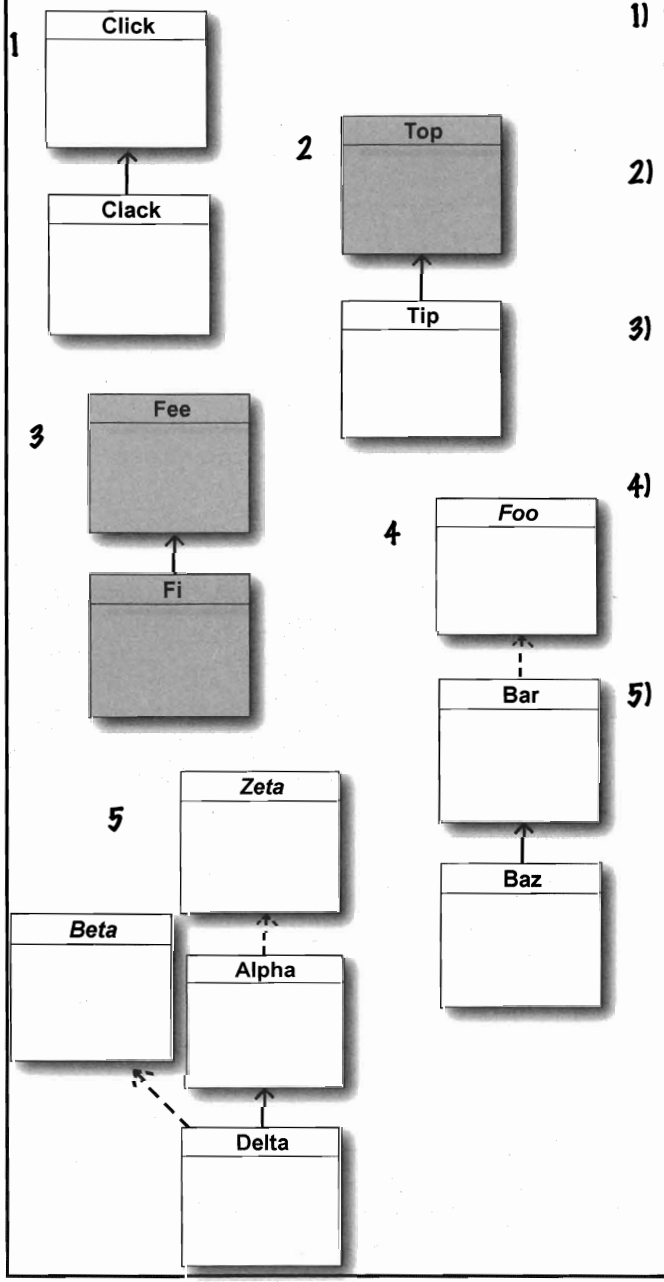
5)



Gyakorlat

Baloldalt osztálydiagramokat találsz. A feladatod az, hogy érvényes Java-deklarációkká alakítsd őket. Az 1-es számút megcsináltuk helyetted (elég nehéz volt).

Az adott diagram:



Hogyan fest a bevezetés?

1) `public class Click { }`

   `public class Clack extends Click { }`

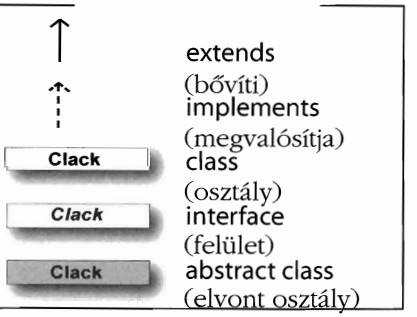
2)

3)

4)

5)

JELÖLÉSEK





Halászd ki a medencéből!



A ***feladatod*** az, hogy kihalászd a kóddarabokat a medencéből, és elhelyezd azokat a kód és a kimenet üresen hagyott helyein. Egy kóddarabot **többször is** felhasználhatsz, viszont nem lesz szükséged minden darabra. A ***cél*** olyan osztályok felépítése, amelyek lefordíthatók, futtathatók, és az alább látható kimenetet eredményezik.

```
_____ Nose {
  _____________________________
}

abstract class Picasso implements _____{
  _____________________________
    return 7;
  }
}

class _____ _____ _____ { }

class _____ _____ _____ {
  _____________________________
    return 5;
  }
}
```

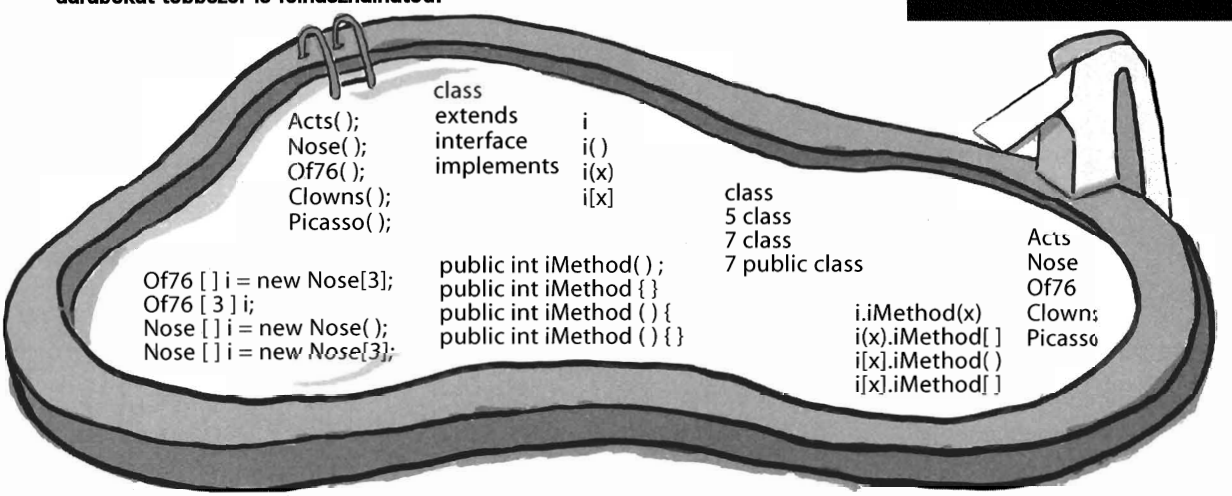
```
public _____ _____ extends Clowns {

  public static void main(String [] args) {
    _____________________________
    i[0] = new _____
    i[1] = new _____
    i[2] = new _____
    for(int x = 0; x < 3; x++) {
      System.out.println(_____
            + " " + _____.getClass( ) );
    }
  }
}
```

Kimenet

```
File  Edit  Window  Help  BeAfraid
%java _____
5 class Acts
7 class Clowns
_____Of76
```

Megjegyzés: A medencében úszó kóddarabokat többször is felhasználhatod!

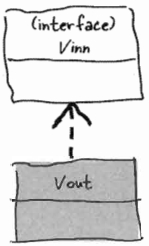




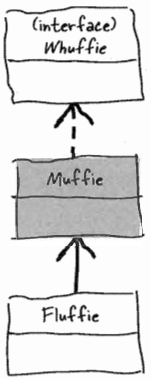
Gyakorlat – Megoldás

Legyen képed hozzá!

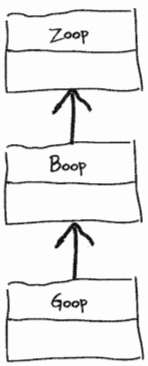
2)



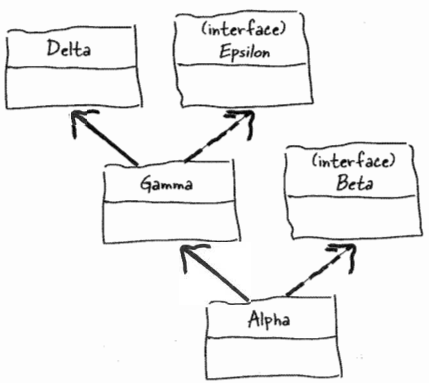
3)



4)



5)



Hogyan fest a bevezetés?

2)
```
public abstract class Top { }
public class Tip extends Top { }
```

3)
```
public abstract class Fee { }
public abstract class Fi extends Fee { }
```

4)
```
public interface Foo { }
public class Bar implements Foo { }
public class Baz extends Bar { }
```

5)
```
public interface Zeta { }
public class Alpha implements Zeta { }
public interface Beta { }
public class Delta extends Alpha implements Beta { }
```



```
interface Nose {
  public int iMethod( );
}
abstract class Picasso implements Nose {
  public int iMethod( ) {
    return 7;
  }
}
class Clowns extends Picasso { }

class Acts extends Picasso {
  public int iMethod( ) {
    return 5;
  }
}
```

```
public class Of76 extends Clowns {
  public static void main(String [] args) {
    Nose [ ] i = new Nose [3];
    i[0] = new Acts( );
    i[1] = new Clowns( );
    i[2] = new Of76( );
    for(int x = 0; x < 3; x++) {
      System.out.println( i [x] . iMethod( )
              + " " + i [x].getClass( ) );
    }
  }
}
```

Kimenet

```
File  Edit  Window  Help  KillTheMime
%java Of76
5 class Acts
7 class Clowns
7 class  Of76
```

Egy objektum élete és halála



Az objektumok születnek és meghalnak. Egy objektum életciklusa a te kezedben van. Te döntöd el, mikor és hogyan **hozod létre**. Te döntöd el, hogy mikor **semmisíted meg**. Eltekintve attól, hogy valójában te magad nem *semmisíted meg* az objektumot, csupán *magára hagyod*. Ha azonban egy objektum egyszer magára maradt, akkor a lelketlen Szemétgyűjtő (Garbage Collector, GC) semmivé porlaszthatja, és visszakövetelheti a memóriát, amelyet az objektum valaha használt. Ha Java-kódot írsz, biztosan létre fogsz hozni objektumokat. Előbb-utóbb pedig el kell engedned közülük néhányat, különben azt kockáztatod, hogy kifogysz a RAM-ból. Ebben a fejezetben azt nézzük meg, hogyan jönnek létre az objektumok, hol laknak, amíg életben vannak, és hogyan tarthatod meg vagy dobhatod el őket hatékonyan. Ez azt jelenti, hogy szóba kerül a kupac, a verem, a hatókör, a konstruktorok (létrehozófüggvények), a szuperkonstruktorok, az üres hivatkozások (null reference), és még sok más minden. Vigyázat: ez a fejezet objektumok halálával kapcsolatos ismereteket tartalmaz, ami megzavarhatja egyesek lelki nyugalmát. Kerüld a túl szoros kötődést.

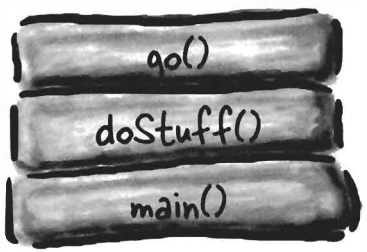
A Verem és a Kupac: ahol az objektumok élnek

Mielőtt megérthetnéd, hogy valójában mi történik, amikor létrehozol egy objektumot, vissza kell lépned egy kicsit. Előbb többet kell tudnod arról, hogy a Javában hol (és meddig) laknak a dolgok – vagyis jobban meg kell ismerned a Vermet (stack) és a Kupacot (heap). A Javában minket, programozókat két memóriaterület érdekel: az a hely, ahol az objektumok élnek (a kupac), és az a hely, ahová a tagfüggvényhívások és a helyi változók kerülnek (a verem). Amikor a Java virtuális gép elindul, kap egy memóriadarabot a háttérben működő operációs rendszertől, amelyet arra használhat, hogy futtassa a Java-programodat. Az, hogy *mennyi* memóriát kap, és hogy a memóriaterületet rugalmasan kezelheted-e, az adott JVM-változattól (és operációs rendszertől) függ, de általában *nem lesz* vele semmi dolgod. És ha jól programozol, valószínűleg nem is fog érdekelni (erre később még visszatérünk).

Azt már tudod, hogy *minden* objektum a szemétgyűjtő kupacban él, de azt még nem néztük meg, hogy hol kapnak helyet a *változók*. Ez pedig attól függ, hogy *milyen fajta* változóról van szó. „Fajta" alatt nem a *típust* (primitív vagy objektumhivatkozás) értjük: az a két *változófajta*, amelynek az élete most érdekes a számunkra, a *példányváltozó* és a *helyi változó*. A helyi változókat *veremváltozóknak* is hívják – ez azért nagy segítséget nyújt ahhoz, hogy kitaláld, hol laknak.

A Verem

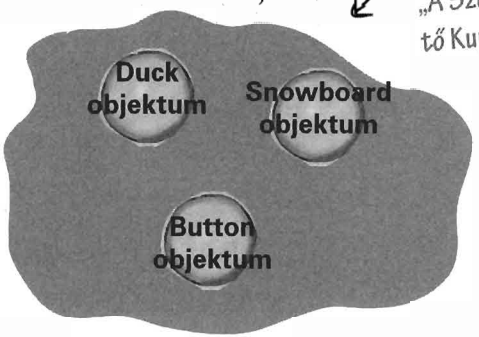
Itt élnek a tagfüggvényhívások és a helyi változók



A Kupac

Itt él ***MINDEN*** objektum

más néven „A Szemétgyűjtő Kupac"



Példányváltozók

A példányváltozókat egy *osztályon*, de nem egy *tagfüggvényen* belül kell bevezetned. A példányváltozók jelképezik azokat a „mezőket", amelyekkel az egyes objektumok rendelkeznek (és amelyeket az osztály minden példányában más-más értékekkel tölthetsz fel). A példányváltozók azon az objektumon belül laknak, amelyhez tartoznak.

```
public class Duck {

    int size;

}
```

Minden Duck objektumnak van „size" példányváltozója.

Helyi változók

A helyi változókat egy *tagfüggvényen* belül kell bevezetned, a tagfüggvény-paramétereket is beleértve. A helyi változók ideiglenesek, és csak addig élnek, amíg a tagfüggvény a veremben van (vagyis amíg a program a tagfüggvény záró kapcsos zárójeléhez nem ér).

```
public void foo(int x) {

    int i = x + 3;

    boolean b = true;

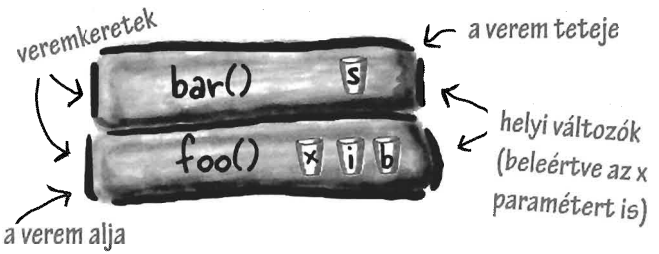
}
```

Az x paraméter, illetve az i és b változók mind helyi változók.

A tagfüggvények egymásra pakolódnak a veremben

Amikor meghívsz egy tagfüggvényt, az a hívásverem tetején köt ki. A verem tetejére valójában egy *keret* (frame) kerül, amely a tagfüggvény állapotát tárolja, beleértve azt is, hogy éppen melyik sora áll végrehajtás alatt, illetve hogy éppen mi a helyi változók értéke. A verem *tetején* levő tagfüggvény mindig az adott verem éppen futó tagfüggvénye. (Egyelőre tegyük fel, hogy csak egy verem létezik. A 14. fejezetben továbbiakat is a programhoz adunk.) Egy tagfüggvény addig marad a veremben, amíg a végrehajtása el nem éri a tagfüggvény záró kapcsos zárójelét (ami azt jelenti, hogy a tagfüggvény végzett a munkájával). Ha a *foo()* tagfüggvény mondjuk meghívja a *bar()* tagfüggvényt, akkor a *bar()* a *foo()* tetejére kerül a veremben.

Hívásverem két tagfüggvénnyel



A verem tetején levő tagfüggvény mindig az éppen végrehajtás alatt álló tagfüggvény.

```
public void doStuff() {
   boolean b = true;
   go(4);
}
public void go(int x) {
   int z = x + 24;
   crazy();
     // képzelj ide még némi kódot
}
public void crazy() {
   char c = 'a';
}
```

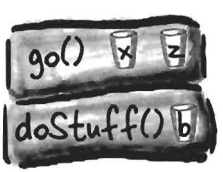
Veremforgatókönyv

A bal oldalon látható kód egy részlet (az osztály többi részével nem foglalkozunk), három tagfüggvénnyel. Az első (*doStuff()*) meghívja a másodikat (*go()*), a második pedig a harmadikat (*crazy()*). Mindegyik tagfüggvény egy helyi változót vezet be a tagfüggvény törzsén belül, a *go()* tagfüggvény pedig egy paraméterváltozót is (ami azt jelenti, hogy a *go()*-nak két helyi változója van).

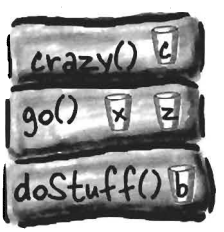
1. Egy másik osztály kódja meghívja a **doStuff()** tagfüggvényt, az pedig egy veremkeretbe kerül a verem tetején. A „**b**” logikai változó a **doStuff()** veremkeretében kap helyet.



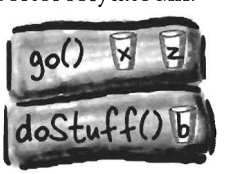
2. A **doStuff()** meghívja a **go()**-t, amelyet a program a verem tetejére *lök* (push). Az „**x**” és „**z**” változók a **go()** veremkeretébe kerülnek.



3. A **go()** meghívja a **crazy()** tagfüggvényt, így most a **crazy()** lesz a verem tetején, a keretében a „**c**” változóval.



4. A **crazy()** végrehajtása befejeződik, így a program *leemeli* (pop) a veremkeretét a veremről. A végrehajtás visszatér a **go()** tagfüggvényhez, és a **crazy()** hívás utáni sortól folytatódik.



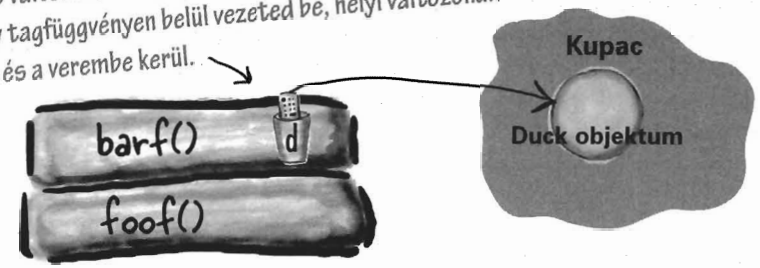
Mi a helyzet azokkal a helyi változókkal, amelyek objektumok?

Emlékezz vissza, hogy a nem primitív változók egy objektumra mutató *hivatkozást* tárolnak, nem magát az objektumot. Azt már tudod, hol laknak az objektumok: a kupacban. Az nem számít, hogy hol vezették be vagy hozták létre őket. Ha ***egy változó egy objektumra mutató hivatkozás, akkor csak a változó (a hivatkozás vagy távirányító) kerül a verembe.***
Maga az objektum továbbra is a kupacba kerül.

```
public class StackRef {
   public void foof() {
      barf();
   }

   public void barf() {
      Duck d = new Duck(24);
   }
}
```

A barf() tagfüggvény bevezeti és létrehozza a „d" nevű új hivatkozó változót, amely egy Duck objektumra mutat. Mivel egy tagfüggvényen belül vezeted be, helyi változónak minősül, és a verembe kerül.



Nem számít, HOL vezetted be az objektumra hivatkozó változót (egy tagfüggvényen belül vagy egy osztály példányváltozójaként) – az objektum mindig mindig mindig a kupacban kap helyet.

Nincsenek
hülye kérdések

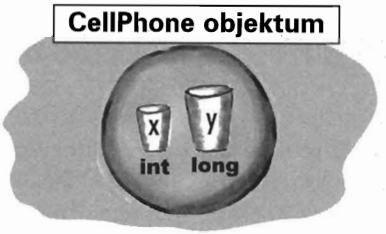
K: Na még egyszer: MIÉRT tanulunk erről a verem–kupac dologról? Miben segít ez nekem? Tényleg tudnom kell róla?

V: A Java-veremmel és -kupaccal kapcsolatos alapismeretek elengedhetetlenek ahhoz, hogy megértsd a változók hatókörét, az objektumok létrehozásával kapcsolatos kérdéseket, a memóriakezelést, a szálakat és a kivételkezelést. A szálakkal és a kivételkezeléssel későbbi fejezetekben foglalkozunk, a többiről azonban még ebben a fejezetben szó lesz. Arról semmit nem kell tudnod, hogy az egyes JVM-ek és rendszerek *hogyan* valósítják meg a vermet és a kupacot. Minden, amit tudnod kell róluk, ezen és az előző oldalon található. Ha az itt olvasottakat megjegyzed, minden olyan témával sokkal, de sokkal könnyebb dolgod lesz, ami ezekre az ismeretekre épül. Még egyszer: egy nap még NAGYON hálás leszel nekünk, hogy lenyomtuk a vermeket és a kupacokat a torkodon.

Újratöltve

- A Java két memóriaterülete érdekes a számodra: a Verem és a Kupac.
- A példányváltozók olyan változók, amelyeket egy osztályon belül, de bármely tagfüggvényen kívül vezettek be.
- A helyi változók olyan változók, amelyeket egy tagfüggvényen vagy tagfüggvény-paraméteren belül vezettek be.
- Minden helyi változó a veremben lakik, annak a tagfüggvénynek megfelelő keretben, amelyben bevezették őket.
- Az objektumra hivatkozó változók ugyanúgy működnek, mint a primitív változók: ha a hivatkozást helyi változóként vezeted be, akkor a verembe kerül.
- Minden objektum a kupacban lakik, függetlenül attól, hogy a hivatkozás helyi vagy példányváltozó-e.

Ha a helyi változók a veremben élnek, akkor hol élnek a példányváltozók?

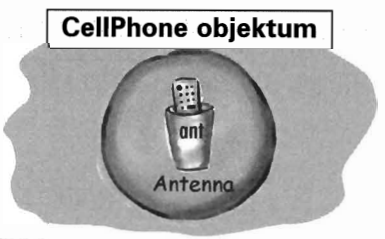


Objektum két primitív példányváltozóval. A változók az objektumban kapnak helyet.

Ha azt mondod, hogy `new CellPhone()`, a Javának helyet kell foglalnia a kupacban a létrehozott CellPhone számára. De *mekkora* helyet? Akkorát, amekkora elég az objektumnak, hogy elférjen benne az összes példányváltozója. Ahogy ebből kitalálhatod, a példányváltozók a kupacban laknak, az objektumuk belsejében.

Emlékezz vissza, hogy az objektumok példányváltozóinak *értékei* az objektum belsejében élnek. Ha az összes példányváltozó primitív típusú, a Java ennek az alaptípusnak megfelelően foglal helyet a példányváltozók számára. Egy int-nek például 32, egy long-nak pedig 64 bitre van szüksége. A Javát nem érdekli, hogy a primitív változókban milyen érték található – egy int típusú változó bitmérete mindig ugyanannyi (32 bit), akár 32, akár 32 000 000 a változó értéke.

De mi a helyzet akkor, ha a példányváltozók *objektumok*? Ha a Mobiltelefonnak (CellPhone) mondjuk van egy (HAS-A) Antennája (vagyis ha a CellPhone objektumnak van egy Antenna típusú hivatkozó változója)?



Objektum egyetlen nem primitív példányváltozóval – egy Antenna objektumra mutató hivatkozással, tényleges Antenna objektum nélkül. Ezt kapod, ha bevezeted a változót, de nem adsz neki kezdőértékül egy valódi Antenna objektumot.

Ha egy új objektumnak olyan példányváltozói vannak, amelyek nem primitívek, hanem objektumhivatkozások, a kérdés valójában a következő: az objektumnak az összes általa hivatkozott objektum számára helyet kell foglalnia? A válasz: *nem igazán.* A Javának mindenképpen helyet kell csinálnia a példányváltozók *értékeinek*, de ne feledd, hogy a hivatkozó változók értéke nem a teljes *objektum*, csupán egy *távirányító* ahhoz. Ha tehát a CellPhone példányváltozóját a nem primitív Antenna típusba tartozóként vezették be, a Java a CellPhone objektumon belül csak az Antenna *távirányítójának* (vagyis a rá hivatkozó változónak), és nem az Antenna *objektumnak* foglal helyet.

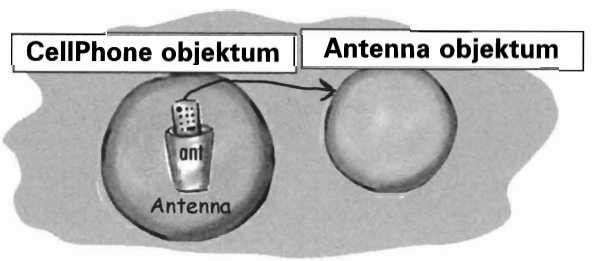
```
public class CellPhone {
  private Antenna ant;
}
```

Jó, de az Antenna *objektum* mikor kap helyet a kupacban? Először is, ki kell derítenünk, hogy maga az Antenna objektum *mikor* jön létre. Ez a példányváltozó bevezetésétől függ. Ha a példányváltozót bevezették, de még nem rendeltek hozzá objektumot, akkor csak a hivatkozó változó (a távirányító) helyét kell lefoglalni:

```
private Antenna ant;
```

A kupacban addig nem jön létre Antenna objektum, amíg a hivatkozó változó értékül nem kap egy új Antenna objektumot:

```
private Antenna ant = new Antenna();
```



Objektum egyetlen nem primitív példányváltozóval, ahol az Antenna változót egy új Antenna objektumhoz rendelted.

```
public class CellPhone {
  private Antenna ant = new Antenna();
}
```

Az objektumszületés csodája

Most, hogy tudod, hol laknak a változók és objektumok, elmerülhetünk az objektumok létrehozásának csodálatos világában. Emlékezz vissza az objektumok bevezetésének és értékkel való ellátásának három lépésére: bevezetsz egy hivatkozó változót, létrehozol egy objektumot, végül hozzárendeled az objektumot a hivatkozáshoz.
Mindezidáig azonban a második lépés – ahol a csoda történik, és „megszületik" az új objektum – Nagy Rejtély maradt. Nos, ideje, hogy megtudd a tényeket az objektumok életéről. *Reméljük, hogy nem vagy ájuldozós fajta.*

Ismételjük át az objektumok bevezetésének, létrehozásának és értékkel való ellátásának 3 lépését:

Létrehozol egy új, osztály vagy felület típusú hivatkozó változót.

1 Hivatkozó változó bevezetése

```
Duck myDuck = new Duck();
```

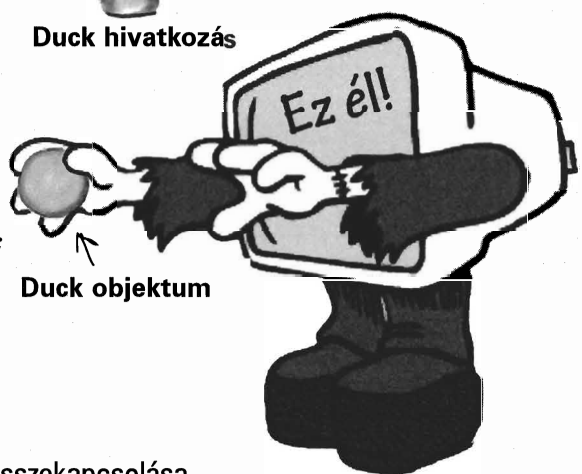


Duck hivatkozás

2 Objektum létrehozása

Itt történik a csoda.

```
Duck myDuck = new Duck();
```



Duck objektum

3 Az objektum és a hivatkozás összekapcsolása

Az új objektumot a hivatkozáshoz rendeled

```
Duck myDuck = new Duck();
```



Duck objektum



Duck hivatkozás

Egy Duck() nevű tagfüggvényt hívunk meg?

Mert úgy néz ki.

A zárójelek miatt úgy tűnik, mintha egy Duck() nevű tagfüggvényt hívnánk.

```
Duck myDuck = new Duck();
```

Nem.
A Duck *konstruktort* hívjuk.

A konstruktorok (létrehozófüggvények) *valóban* úgy néznek ki, mint a tagfüggvények, de nem azok. Az a kód található bennük, ami akkor fut le, amikor azt mondod, hogy **new**. Más szavakkal, *az a kód, aminek a végrehajtására akkor kerül sor, amikor példányosítasz egy objektumot.*

A konstruktorok meghívásának egyetlen módja a **new** kulcsszó és utána az osztály nevének használata. A JVM megkeresi a megadott osztályt, és meghívja annak konstruktorát. (Na jó, elméletileg nem ez az *egyetlen* módja annak, hogy meghívj egy konstruktort, egy konstruktoron *kívülről* azonban nem lehet másképp. Bizonyos korlátok között egy másik konstruktorból is meghívhatsz egy konstruktort, de ezzel majd a fejezet egy későbbi részében foglalkozunk.)

A konstruktorban található az a kód, ami akkor fut le, amikor példányosítasz egy objektumot, vagyis amikor egy osztálytípusra azt mondod, hogy `new`.

Minden osztálynak, amelyet létrehozol, van konstruktora, még akkor is, ha nem magad írod meg azt.

De hol van a konstruktor?
Ha nem mi írtuk meg, akkor ki?

Te is írhatsz konstruktort egy osztály számára (hamarosan sort is kerítünk rá), de ha nem teszed meg, a ***fordítóprogram megteszi helyetted***. Így fest a fordító alapértelmezett konstruktora:

```
public  Duck() {

}
```

Észrevetted, hogy valami hiányzik? Miben különbözik ez egy tagfüggvénytől?

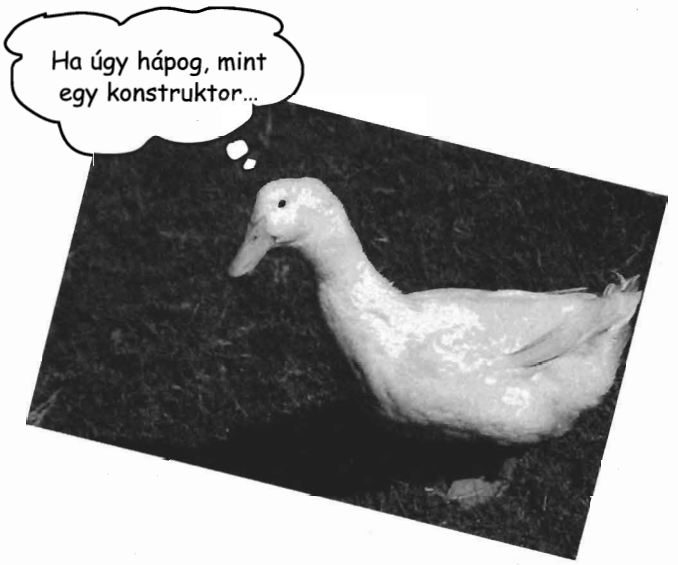
A neve ugyanaz, mint az osztálynak. Ez kötelező.

```
public Duck() {
    // ide jön a konstruktor kódja
}
```

Hol van a visszatérési típus? Ha ez tagfüggvény lenne, szükség lenne egy visszatérési típusra a „public" és a „Duck" között.

Hozz létre egy Kacsát!

A konstruktorok egyik legfontosabb vonása, hogy az *előtt* futnak le, hogy az objektumot hozzárendelhetnéd egy hivatkozáshoz. Ez azt jelenti, hogy lehetőséged van közbeavatkozni, és felkészíteni a használatra az objektumot. Más szavakkal, mielőtt bárki használhatná az objektum távirányítóját, az objektumnak esélye nyílik, hogy segítsen önmaga létrehozásában. A Duck (Kacsa) konstruktorunkban nem csinálunk semmi hasznosat, az eseménysort azonban jól szemlélteti:




```
public class Duck {

   public Duck() {
      System.out.println("Quack");
   }
}
```

Konstruktorkód

A konstruktor lehetőséget ad rá, hogy beavatkozz a `new` utasításba.

```
public class UseADuck {

   public static void main (String[] args) {
      Duck d = new Duck();
   }
}
```

Ez hívja meg a Duck konstruktort.

```
File Edit Window Help Quack
% java UseADuck

Quack
```

Hegyezd ki a ceruzád!

A konstruktor lehetővé teszi, hogy fejest ugorj az objektum létrehozásának közepébe – a **new** utasításba. El tudsz képzelni olyan helyzeteket, amikor ez hasznos lehet? Az itt felsorolt műveletek közül melyiknek vennéd hasznát egy Car (Autó) osztály konstruktorában, ha a Car egy Racing Game (Autóverseny) része? Pipáld ki azokat a műveleteket, amelyekhez el tudsz képzelni egy forgatókönyvet!

- ❑ Növeled egy számláló értékét, hogy nyomon követhesd, hány objektum jött létre az adott osztálytípusból.
- ❑ Egy változóban rögzíted a futáskori állapotot (adatokat arról, hogy mi történik MOST).
- ❑ Értéket adsz az objektum fontos példányváltozóinak.
- ❑ Kiolvasol és mentesz egy hivatkozást arra az objektumra, amelyik éppen *létrehozza* az új objektumot.
- ❑ Az objektumot egy tömblistához (ArrayList) adod.
- ❑ Tartalmazott (HAS-A) objektumokat hozol létre.
- ❑ _____ (ide találj ki valamit te)

Új Kacsa kezdőállapotának meghatározása

A legtöbben arra használják a konstruktorokat, hogy meghatározzák egy objektum kezdőállapotát, vagyis hogy értéket adjanak az objektum példányváltozóinak.

```
public Duck() {
   size = 34;
}
```

Ez szép és jó, ha a Duck osztály *fejlesztője* tudja, hogy mekkorának kell lennie a Duck objektumnak. De mi a helyzet akkor, ha a Duck-ot *használó* programozónak szeretnénk lehetővé tenni, hogy eldöntse, milyen nagy legyen egy adott Kacsa?
Tegyük fel, hogy a Duck objektumnak van egy size (méret) példányváltozója, és azt szeretnéd, hogy a Duck osztályt használó programozó beállíthassa az új Kacsák méretét. Hogyan érheted el ezt?
Nos, az osztályhoz adhatsz például egy setSize() – „beallitMeret" – nevű beállító tagfüggvényt, de ez a Duck-ot átmenetileg méret nélkül hagyná*, és arra kényszerítené a Duck felhasználóját, hogy *két* utasítást írjon – egyet a Duck létrehozásához, és egyet a setSize() tagfüggvény meghívásához. Az alábbi kód egy beállító tagfüggvény segítségével állítja be az új Kacsa kezdeti méretét:

```
public class Duck {
   int size;                                  ← példányváltozó

   public Duck() {
      System.out.println("Quack");            ← konstruktor
   }

   public void setSize(int newSize) {         ← beállító tagfüggvény
      size = newSize;
   }
}
```

```
public class UseADuck {

   public static void main (String[] args){
      Duck d = new Duck();
                                  ← Itt valami nem jó. A Kacsa a kódnak ezen a pontján életben van, de nincs mérete!* Ezek után pedig arra támaszkodsz, hogy a kacsahasználó TUDJA, hogy a Kacsa létrehozása kétrészes eljárás, amelyben először a konstruktort, majd a beállító tagfüggvényt kell meghívnia.
      d.setSize(42);
   }
}
```

*A példányváltozóknak van alapértelmezett értékük: a számértékű alaptípusok esetében 0 vagy 0.0, a logikai típusoknál a false, a hivatkozások esetében pedig a null.

Nincsenek hülye kérdések

K: Miért kell konstruktort írnom, ha egyszer a fordító ezt megteszi helyettem?

V: Ha az objektum kezdőértékekkel való ellátását és használatra való előkészítését segítő kódra van szükséged, saját konstruktort kell írnod. Előfordulhat például, hogy adatokat kell kapnod a felhasználótól, mielőtt befejezhetnéd az objektum előkészítését. Ezen kívül van még egy ok, amiért esetleg konstruktort kell írnod, még ha magadnak nincs is szükséged konstruktorkódra. A dolog az alaposztály konstruktorával kapcsolatos, és hamarosan bővebben beszélünk róla.

K: Hogyan különböztethetek meg egy konstruktort egy tagfüggvénytől? Lehet olyan tagfüggvényem is, aminek ugyanaz a neve, mint az osztálynak?

V: A Java megengedi, hogy az osztály nevével megegyező nevű tagfüggvényt vezess be – ettől az még nem lesz konstruktor. A tagfüggvényeket a visszatérési típus különbözteti meg a konstruktoroktól. A tagfüggvényeknek rendelkezniük *kell* visszatérési típussal, a konstruktoroknak azonban *nem lehet* visszatérési típusuk.

K: A konstruktorok öröklődnek? Ha nem adok meg konstruktort, de az alaposztály igen, az alapértelmezett helyett az alaposztály konstruktora lép működésbe?

V: Nem. A konstruktorok nem öröklődnek. Hamarosan erre is kitérünk.

A Kacsa fontos állapotjellemzőinek*

meghatározása a konstruktor segítségével



Ha egy objektumot nem szabad addig használatba venni, amíg egy vagy több állapotjellemzője (példányváltozója) kezdőértéket nem kapott, ne engedd senkinek, hogy rátegye a kezét, amíg be nem fejezted az előkészítést! Például általában nagyon kockázatos hagyni, hogy valaki létrehozzon egy új Duck objektumot – és hivatkozzon rá –, mert az addig nem áll készen teljesen a használatra, amíg valaki meg nem hívja a *setSize()* tagfüggvényét. De honna *tudja* egyáltalán a Duck-felhasználó, hogy egy új Kacsa létrehozása után meg kell hívnia a beállító tagfüggvényt? Az előkészítő kódot a legjobb a konstruktorban elhelyezni. Ehhez csak annyit kell tenned, hogy argumentumokkal látod el a konstruktort.

```
public class Duck {
   int size;

   public Duck(int duckSize) {
      System.out.println("Quack");

      size = duckSize;

      System.out.println("size is " + size);
   }
}
```

Egy int paramétert adunk a Duck konstruktorhoz.

Az argumentum értékével állítjuk be a size példányváltozót.

```
public class UseADuck {

  public static void main (String[] args) {
     Duck d = new Duck(42);
  }
}
```

Ezúttal csak egy utasítás van. Egyetlen utasítással hozzuk létre az új Kacsát, és állítjuk be a méretét.

Átadunk egy értéket a konstruktornak.

```
File Edit Window Help Honk
% java UseADuck
Quack
size is 42
```

*Ezzel nem arra célzunk, hogy a Kacsának nem minden állapotjellemzője lényegtelen.

Könnyítsd meg a Kacsa létrehozását!

Alkalmazz argumentum nélküli konstruktort!

Mi történik, ha a Duck konstruktora argumentumot vár? Gondold végig! Az előző oldalon csak *egy* Duck konstruktor volt, amely egy int típusú argumentumot várt a Kacsa *méretének* beállításához. Ez nem tűnik nagy ügynek, de egy programozónak megnehezíti, hogy új Duck objektumot hozzon létre, különösen ha a programozó nem *tudja*, hogy mekkorának kell lennie egy Kacsának. Nem lenne nagy segítség, ha a Kacsáknak lenne egy alapértelmezett méretük, hogy a felhasználók akkor is létrehozhassanak működő Kacsákat, ha nem tudják a pontos méretüket?

Tegyük fel, hogy azt szeretnéd, hogy a Duck-felhasználóknak KÉT választási lehetőségük legyen egy Kacsa létrehozására: az egyik esetben megadnák a Kacsa méretét (a konstruktor argumentumaként), míg a másikban nem adnának meg méretet, és így egy alapértelmezett méretű Kacsát kapnának.

Ezt nem érheted el tisztán egyetlen konstruktorral. Emlékezz vissza, hogy ha egy tagfüggvénynek (vagy konstruktornak – ugyanazok a szabályok érvényesek rá) paramétere van, akkor *kötelező* átadnod egy megfelelő argumentumot, amikor meghívod a függvényt. Nem mondhatod egyszerűen azt, hogy „ha valaki nem ad át semmit a konstruktornak, akkor használd az alapértelmezett méretet", mert a programot még lefordítani sem sikerül, ha a konstruktorhívásban nem szerepel egy int argumentum. Valami ilyesmit persze *megtehetnél*:

```
public class Duck {
   int size;

   public Duck(int newSize) {
      if (newSize == 0) {
         size = 27;
      } else {
         size = newSize;
      }
   }
}
```

Ha a paraméter értéke nulla, akkor a Kacsa az alapértelmezett méretet kapja, egyébként pedig a paraméter értéke határozza meg a méretét. NEM túl jó megoldás.

Ez azonban elég esetlen megoldás, az új Duck objektumot létrehozó programozónak ugyanis *tudnia* kell, hogy az alapértelmezett kacsaméretet a „0" átadásával kaphatja meg. Elég szörnyű. Mi van, ha egy másik programozó nem tudja ezt? Vagy ha *tényleg* nulla méretű kacsát akar? (Feltételezve, hogy ez egyáltalán megengedett. Ha nem akarsz nulla méretű Duck objektumokat, akkor tegyél ellenőrző kódot a konstruktorba.) Arra célzunk, hogy nem mindig lehet különbséget tenni a „Nulla méretet akarok" és a „Nullát adok át, hogy az alapértelmezett méretet használd, bármekkora legyen is az" konstruktor-argumentum között.

Valójában KÉT módszert szeretnél az új Duck objektumok létrehozására:

```
public class Duck2 {
   int size;

   public Duck2() {
      // az alapértelmezett méret megadása
      size = 27;
   }

   public Duck2(int duckSize) {
      // a duckSize paraméter használata
      size = duckSize;
   }
}
```

Így hozhatsz létre egy Kacsát, ha tudod a méretét:

```
Duck2 d = new Duck2(15);
```

És így hozhatod létre akkor, ha a mérete *ismeretlen* a számodra:

```
Duck2 d2 = new Duck2();
```

A „kétféleképpen csinálhatsz Kacsát" elv tehát *két* konstruktort igényel. Az egyik egy int argumentumot vár, míg a másiknak nem kell argumentum. Ha ***egy osztályban egynél több konstruktor található, az azt jelenti, hogy túlterhelt konstruktorokkal rendelkezel.***

A fordító nem mindig hoz létre egy argumentum nélküli konstruktort a számomra?

Nem!



Azt hiheted, hogy ha csak az argumentumokat váró konstruktort írod meg, akkor a fordító látni fogja, hogy nincs argumentum nélküli konstruktorod, és beszúr egyet helyetted. A dolog azonban nem így működik. A fordító *csak akkor* vesződik a konstruktorok létrehozásával, *ha semmit sem mondasz róluk.*

Ha írsz egy argumentumokat váró konstruktort, de argumentum nélküli konstruktort *is* szeretnél, akkor azt is neked kell megírnod!

Amint megadsz egy *saját* konstruktort – BÁRMILYET –, a fordító visszavonul, és azt mondja: „Oké, haver, most már te felelsz a konstruktorokért".

Ha egy osztályban több konstruktor található, akkor a konstruktorok argumentumlistájának különböznie KELL.

Az argumentumlistába az argumentumok sorrendje és típusa tartozik bele. Amíg ezek különböznek, több konstruktorod is lehet. Ez a tagfüggvényekre is érvényes, de ezzel egy másik fejezetben foglalkozunk.

Ha túlterhelt konstruktoraid vannak, az azt jelenti, hogy az osztályban több konstruktor található.

A fordítás csak akkor sikerül, ha minden konstruktor *különböző* argumentumlistával rendelkezik.

Az alábbi osztály érvényes, mert mind az öt konstruktornak különböző az argumentumlistája. Ha két konstruktorod is csak egy int értéket várna, az osztályt nem tudnád lefordíttatni. A paraméterváltozó elnevezése nem számít, csak a változó *típusa* (int, Dog stb.) és a változók *sorrendje*. Két azonos paramétertípusokkal rendelkező konstruktorod tehát *lehet*, ***amennyiben a paraméterek sorrendje eltér***. Egy String, majd egy int értéket váró konstruktor nem ugyanaz, mint egy int, majd egy String értéket váró konstruktor.

Az öt különböző konstruktor azt jelenti, hogy öt különböző módon hozhatsz létre egy gombát (mushroom).



```
public class Mushroom {

  public Mushroom(int size) { }

  public Mushroom( ) { }

  public Mushroom(boolean isMagic) { }

  public Mushroom(boolean isMagic, int size) { }

  public Mushroom(int size, boolean isMagic) { }
}
```

ha tudod a méretet, de azt nem, hogy varázslat-e

ha nem tudsz semmit

ha tudod, hogy varázslat-e vagy sem, de nem tudod a méretet

ez a kettő ugyanazokkal az argumentumokkal rendelkezik, de más sorrendben, ezért ez oké

ha tudod, hogy varázslat-e vagy sem, ÉS a méretet is ismered

Újratöltve

- A példányváltozók az objektumuk belsejében élnek, a kupacban.
- Ha a példányváltozó egy objektumra mutató hivatkozás, akkor a hivatkozás és a hivatkozott objektum is a kupacban található.
- A konstruktor az a kód, ami akkor fut le, amikor egy osztálytípusra azt mondod, hogy **new**.
- A konstruktor nevének meg kell egyeznie az osztályéval, és *nem lehet* visszatérési típusa.
- A konstruktort arra használhatod, hogy megadd a létrehozni kívánt objektum kezdőállapotának jellemzőit (vagyis a példányváltozók értékét).
- Ha egy osztályba nem teszel konstruktort, akkor a fordító beszúr egy alapértelmezettet.
- Az alapértelmezett konstruktor mindig argumentum nélküli.
- Ha egy osztályban elhelyezel egy konstruktort – bármilyet –, a fordító nem hoz létre alapértelmezett konstruktort.
- Ha argumentumok nélküli konstruktort szeretnél, és már tettél az osztályba egy argumentumokat váró változatot, akkor az argumentum nélküli konstruktort is magadnak kell megírnod.
- Mindig adj meg argumentum nélküli konstruktort, amikor csak teheted, hogy megkönnyítsd a programozóknak a működőképes objektumok létrehozását. Adj meg alapértelmezett értékeket is.
- A túlterhelt konstruktor azt jelenti, hogy az osztályban egynél több konstruktor található.
- A túlterhelt konstruktoroknak eltérő argumentumlistával kell rendelkezniük.
- Egy osztályban nem lehet két, azonos argumentumlistával rendelkező konstruktorod. Az argumentumlistát az argumentumok sorrendje és típusa egyaránt meghatározza.
- A példányváltozók akkor is kapnak alapértelmezett értéket, ha kifejezetten nem rendelsz hozzájuk ilyet. Az alapértelmezett érték a primitívek esetében 0., 0.0, iletve false, míg a hivatkozások esetében null.

Hegyezd ki a ceruzád!



Párosítsd össze az egyes **new Duck()** hívásokat azzal a konstruktorral, ami akkor fut le, amikor létrehozod az adott Duck-példányt! A legkönnyebb párosítást segítségképpen elvégeztük helyetted.

```
public class TestDuck {

  public static void main(String[] args){

    int weight = 8;
    float density = 2.3F;
    String name = "Donald";
    long[] feathers = {1,2,3,4,5,6};
    boolean canFly = true;
    int airspeed = 22;

    Duck[] d = new Duck[7];

    d[0] = new Duck();

    d[1] = new Duck(density, weight);

    d[2] = new Duck(name, feathers);

    d[3] = new Duck(canFly);

    d[4] = new Duck(3.3F, airspeed);

    d[5] = new Duck(false);

    d[6] = new Duck(airspeed, density);
  }
}
```

```
class Duck {

  int pounds = 6;
  float floatability = 2.1F;
  String name = "Generic";
  long[] feathers = {1,2,3,4,5,6,7};
  boolean canFly = true;
  int maxSpeed = 25;

  public Duck() {
    System.out.println("type 1 duck");
  }

  public Duck(boolean fly) {
    canFly = fly;
    System.out.println("type 2 duck");
  }

  public Duck(String n, long[] f) {
    name = n;
    feathers = f;
    System.out.println("type 3 duck");
  }

  public Duck(int w, float f) {
    pounds = w;
    floatability = f;
    System.out.println("type 4 duck");
  }

  public Duck(float density, int max) {
    floatability = density;
    maxSpeed = max;
    System.out.println("type 5 duck");
  }
}
```

K: Korábban azt mondtátok, hogy nem árt, ha van argumentum nélküli konstruktor, hogy ha valaki azt hívja meg, alapértelmezett értéket adhassunk a „hiányzó" argumentumoknak. Néha azonban lehetetlen alapértelmezést megadni, nem? Előfordulhat, hogy jobb, ha az osztályban nincs argumentum nélküli konstruktor?

V: Igazad van: előfordulhat, hogy egy argumentum nélküli konstruktornak nincs értelme. Ilyesmit látni fogsz majd a Java API-ban, ahol egyes osztályok nem rendelkeznek argumentum nélküli konstruktorral. A Color (Szín) osztály például egy…hm…színt ábrázol. A színobjektumokat többek között arra használhatod, hogy beállítsd vagy megváltoztasd a betűk vagy egy gomb színét a képernyőn. Amikor létrehozol egy Color-példányt, az egy adott színt fog jelképezni (tudod, „haláli csokoládébarna", „kékhalál-kék", „botrányos vörös" stb.). Ha színobjektumot hozol létre, valahogy meg kell adnod a színt:

```
Color c = new Color(3,45,200);
```

(Itt három int értékkel adjuk meg az RGB – Red, Green, Blue, vagyis Vörös, Zöld és Kék – színösszetevőket. A színek használatában később merülünk el, amikor a Swingről tanulunk.) Ha nem adnád meg a színt, mit kapnál? A Java API programozói dönthettek *volna* úgy, hogy ha argumentum nélkül hívod meg a Color konstruktort, akkor a mályva egy csábos árnyalatát kapd, de győzött a jó ízlés. Ha argumentum nélkül próbálsz színt létrehozni…

```
Color c = new Color();
```

```
File Edit Window Help StopBeingStupid
cannot resolve symbol
:constructor Color()
location: class
java.awt.Color
Color c = new Color();
          ^
1 error
```

A fordító kiakad, mert nem talál argumentum nélküli konstruktort a Color osztályban.

Nanoismétlés: négy megjegyzendő dolog a konstruktorokkal kapcsolatban

1. A konstruktor az a kód, ami akkor fut le, amikor valaki egy osztálytípusra azt mondja, hogy **new**.

```
Duck d = new Duck();
```

2. A konstruktor nevének meg kell egyeznie az osztályáéval, és ***nem lehet*** visszatérési típusa.

```
public Duck(int size) { }
```

3. Ha egy osztályba nem teszel konstruktort, akkor a fordító beszúr egy alapértelmezett konstruktort helyetted. Az alapértelmezett konstruktor mindig argumentum nélküli.

```
public Duck() { }
```

4. Egy osztályban egynél több konstruktor is lehet, feltéve, hogy az argumentumlistájuk különbözik. Ha egy osztályban több konstruktor található, az azt jelenti, hogy túlterhelt konstruktorokkal rendelkezel.

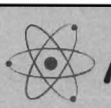
```
public Duck() { }

public Duck(int size) { }

public Duck(String name) { }

public Duck(String name, int size) { }
```

A felmérések szerint az összes agytornáztató gyakorlat elvégzése 42%-kal növeli az agysejtek méretét. És ismered a mondást: „nagy agysejt,...".



Agytorna

Mi a helyzet az alaposztályokkal?

Amikor létrehozol egy Dog-ot, akkor a Canine konstruktorának is le kell futnia?

Ha az alaposztály elvont, akkor is *kell* konstruktorának lennie?

Ezekre a kérdésekre a következő néhány oldalon kapod meg a választ, ezért most állj meg, és gondolkodj el a konstruktorok és alaposztályok kapcsolatán.

Nincsenek

hülye kérdések

K: A konstruktoroknak nyilvánosaknak kell lenniük?

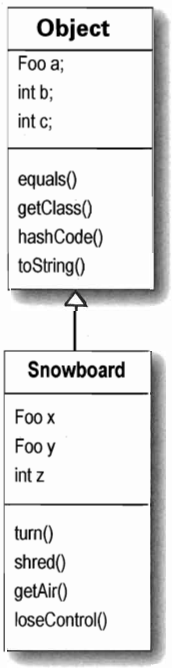
V: Nem. Egy konstruktor lehet **public**, **private**, de alapértelmezett (*default*) is (ami azt jelenti, hogy nem áll előtte semmilyen módosító). Az *alapértelmezett* hozzáférésről a 16. fejezetben és a B függelékben lesz bővebben szó.

K: Milyen haszna lehet egy privát konstruktornak? Hiszen senki nem hívhatja meg, így nem is hozhat létre új objektumot az adott típusból!

V: Ez nem egészen igaz. Ha valamit privátként jelölsz meg, az nem azt jelenti, hogy *senki* nem férhet hozzá, csak azt, hogy az *osztályon kívülről* nem elérhető. Most biztos a 22-es csapdája jut az eszedbe: csak a privát konstruktorral rendelkező osztályban levő kód hozhat létre új objektumot az osztályból, de hogyan futtathatsz kódot az osztályból, ha előbb létre kellene hoznod belőle egy objektumot? Hogyan férhetsz hozzá bármihez az osztályból? *Türelem, ifjú padawan.* A következő fejezetből megtudod.

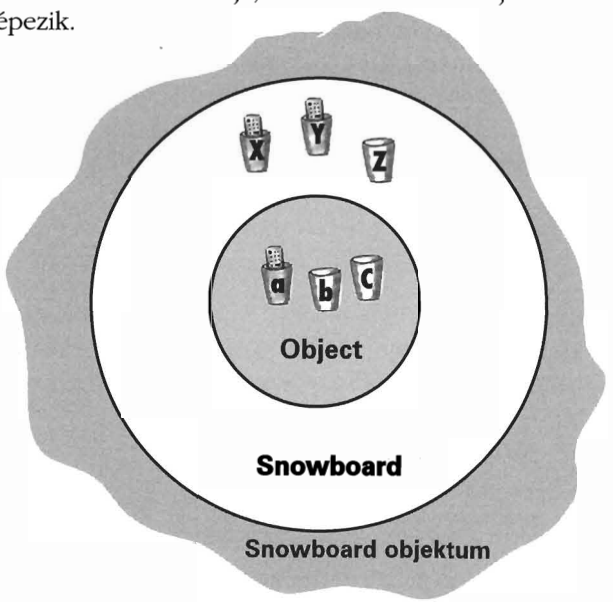
Várjunk csak egy percet... eddig egy szót sem ejtettünk az alaposztályok és az öröklés, illetve a konstruktorok kapcsolatáról!

Itt válik izgalmassá a dolog. Emlékszel arra a részre az előző fejezetből, amikor megmutattuk, hogyan veszi körbe a Snowboard objektum a Snowboard osztály Object részét jelképező belső magot? A Lényeg az volt, hogy minden objektumra érvényes, hogy nem csak a *saját* bejelentett példányváltozóit tárolja, hanem *minden olyasmit is, ami az ősosztályaiból származik* (ami legalább az Object osztályt jelenti, hiszen minden osztály bővítése az Object-nek).
Amikor tehát létrejön egy objektum (mert valaki azt mondta, hogy **new** – egy objektum létrehozásának nincs ***semmilyen más módja***, mint hogy valahol, valaki **new**-t mond az osztálytípusra), az objektum *minden* példányváltozója számára helyet kap, egészen az öröklési fa tetejéig. Gondolkodj el ezen egy percre...
Egy alaposztálynak lehetnek beállító tagfüggvényei, amelyek betokoznak egy privát változót, de az említett változónak *valahol* élnie kell. Amikor egy objektum létrejön, az majdnem olyan, mintha *több* objektum öltene anyagi formát – a „new"-zott objektum, illetve alaposztályonként egy-egy további objektum. Elméletileg persze könnyebb úgy elképzelni, mint ahogy az alábbi ábra mutatja, ahol a létrehozott objektum *rétegekből* áll, amelyek az egyes alaposztályokat jelképezik.



Az Object-nek elérőfüggvényekbe csomagolt példányváltozói vannak. Ezek a példányváltozók akkor jönnek létre, amikor valamelyik alosztályt példányosítják. (Ezek nem az Object IGAZI változói, de ez minket nem izgat, hiszen be vannak tokozva.)

A Snowboard-nak is vannak saját példányváltozói, ezért egy Snowboard objektum létrehozásához mindkét osztály példányváltozói számára helyet kell csinálnunk.



Egyetlen objektum a kupacban.
A kupacban itt csak EGY objektum – egy Snowboard objektum – található. Az objektum azonban önmagából mind a Snowboard, mind az Object részt tartalmazza. Mindkét osztály összes példányváltozójának itt kell helyet kapnia.

Az alaposztályok konstruktorainak szerepe egy objektum életében

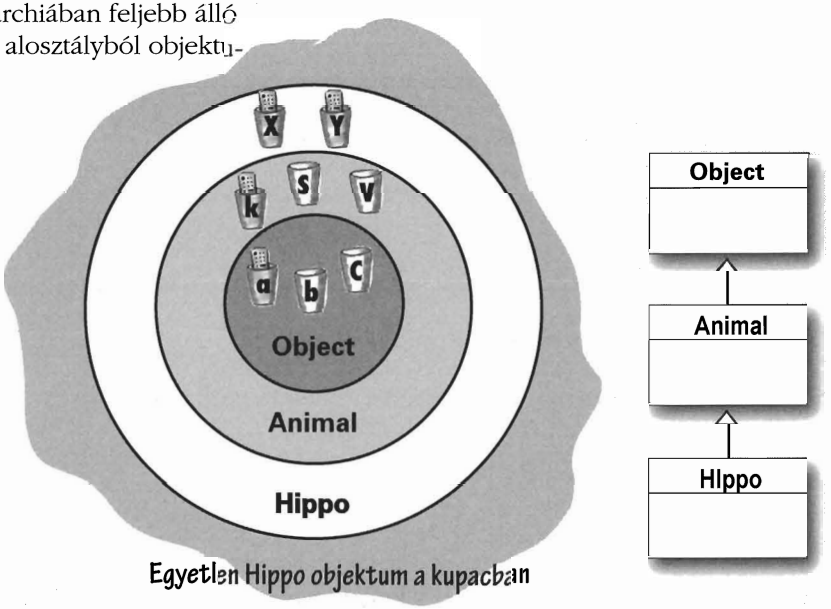
Amikor létrehozol egy új objektumot, az objektum öröklési fájában található összes konstruktornak le kell futnia.

Ezt jegyezd meg jól.

Ez azt jelenti, hogy minden alaposztálynak van konstruktora (mivel minden osztálynak van), és a hierarchiában feljebb álló valamennyi konstruktor lefut, amikor egy alosztályból objektumot hoznak létre.

A **new** kimondása Fontos Esemény. Ez indítja el a konstruktorok láncreakcióját. És igen, még az elvont osztályok is rendelkeznek konstruktorral. Bár egy elvont osztályra soha mondhatsz new-t, az attól még alaposztály, ezért a konstruktora minden alkalommal lefut, amikor valaki példányt hoz létre egy konkrét alosztályából.

Az őskonstruktorok azért futnak le, hogy felépítsék az objektumnak az alaposztályokból származó részét. Emlékezz vissza, hogy egy alosztály olyan tagfüggvényeket is örökölhet, amelyek az ősosztály állapotától (vagyis az alaposztály példányváltozóinak értékétől) függnek. Ahhoz, hogy egy objektum teljesen alakot öltsön, az alaposztályokból származó részeit is teljesen fel kell építeni – ezért *muszáj*, hogy lefussanak az őskonstruktorok.



Egyetlen Hippo objektum a kupacban

Az öröklési fa valamennyi osztályának összes példányváltozóját be kell vezetni és kezdőértékkel ellátni. Még ha az Animal osztálynak vannak is olyan példányváltozói, amelyeket a Hippo osztály nem örököl (például mert az említett változók privát jelzésűek), a Hippo akkor is olyan Animal-tagfüggvényekre támaszkodik, amelyek *használják* ezeket a változókat.

Amikor egy konstruktor lefut, azonnal meghívja az ősosztálybeli konstruktorait, a láncban egészen addig haladva felfelé, amíg el nem ér az Object osztály konstruktoráig.

A következő néhány oldalon megtanulod, hogyan történik az alaposztálybeli konstruktorok meghívása, és hogyan hívhatod meg őket te magad. Ezenkívül azt is megtudod, mit kell tenned, ha az alaposztálybeli konstruktor argumentumokat vár.

A **Hippo** objektum egyben egyfajta (IS-A) **Animal**, és egyfajta **Object**. Ha **Hippo**-t akarsz létrehozni, akkor a **Hippo**-nak az **Animal** és az **Object** részét is fel kell építened.

Mindez abban az eljárásban megy végbe, amelyet **konstruktorláncolásnak** hívnak.

Egy Vízilovat csak úgy hozhatsz létre, hogy felépíted az Állat és Objektum részeit is...

```
public class Animal {
   public Animal() {
      System.out.println("Making an Animal");
   }
}
```

```
public class Hippo extends Animal {
   public Hippo() {
      System.out.println("Making a Hippo");
   }
}
```

```
public class TestHippo {
   public static void main (String[] args) {
      System.out.println("Starting...");
      Hippo h = new Hippo();
   }
}
```

Hegyezd ki a ceruzád!

Melyik az igazi kimenet? Mit ír ki a baloldalt látható kód, ha lefuttatod a TestHippo osztályt? Az A vagy a B kimenetet?
(a választ a lap alján találod)

A

```
File Edit Window Help Swear
% java TestHippo
Starting...
Making an Animal
Making a Hippo
```

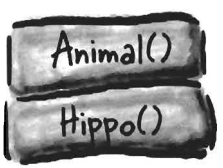
B

```
File Edit Window Help Swear
% java TestHippo
Starting...
Making a Hippo
Making an Animal
```

1. Egy másik osztály kódja azt mondja, hogy `new Hippo()`, a **Hippo()** konstruktor pedig egy veremkeretbe kerül a verem tetején.



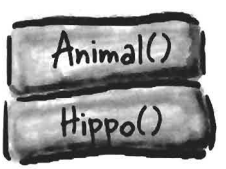
2. A **Hippo()** meghívja az alaposztálya konstruktorát, ami az **Animal()** konstruktort helyezi a verem tetejére.



3. Az **Animal()** meghívja az alaposztálya konstruktorát, ami az **Object()** konstruktort löki a verem tetejére, mivel az Object az Animal alaposztálya.



4. Az **Object()** befejezi a feladatát, így a veremkerete *lekerül* a veremről. A végrehajtás visszakerül az **Animal()** konstruktorhoz, és attól a sortól folytatódik, amelyik előtt az Animal meghívta az alaposztálya konstruktorát.



Az első (A). Először a Hippo() konstruktort hívjuk meg, de az Animal konstruktora fejezi be előbb a működését.

Hogyan hívhatod meg egy alaposztály konstruktorát?

Azt gondolhatod, hogy ha a Duck az Animal-t bővíti, akkor valahol – mondjuk egy Duck konstruktorban – az Animal()-t hívod meg. A dolog azonban nem így működik:

```
public class Duck extends Animal {
   int size;

   public Duck(int newSize) {
      Animal();
      size = newSize;
   }
}
```

ROSSZ!

NEM! Ez nem megengedett!

Egy alaposztálybeli konstruktort kizárólag a ***super()*** hívással lehet meghívni. Így van – a „***szuperkonstruktort***" (őskonstruktort, alaposztálybeli konstruktort) a ***super()*** hívja meg.
Ki hitte volna?

```
public class Duck extends Animal {
   int size;

   public Duck(int newSize) {
      super();
      size = newSize;
   }
}
```

csak mondd azt, hogy super()

A *super()* hívás a konstruktorban az alaposztály konstruktorát helyezi a Verem tetejére. És mit gondolsz, mit csinál a szuperkonstruktor? *Meghívja a saját alaposztályának a konstruktorát.* És ez így megy, egészen addig, amíg a Verem tetejére az Object konstruktora nem kerül. Miután az *Object()* elvégezte a feladatát, leugrik a verem tetejéről, így a verem tetején most már az alatta levő elem (az *Object()*-et meghívó alosztálybeli konstruktor) lesz.
Ez a konstruktor szintén befejezi a feladatát, és ez addig folytatódik, amíg a verem tetejére az eredeti konstruktor nem kerül, hogy *az* is befejezhesse a munkáját.

De akkor hogyan lehetséges, hogy eddig elvoltunk a super() hívás nélkül?

Ezt már biztos kitaláltad.

Jó barátod, a fordító, beszúrja a super() hívást, ha te nem teszed meg.

A fordító tehát *kétféle* módon vesz részt a konstruktorok létrehozásában:

1. **Ha *nem* adsz meg konstruktort:**
   A fordító beszúr egyet, ami valahogy így néz ki:
   ```
   public OsztályNév() {
   super();
   }
   ```

2. **Ha *megadsz* egy konstruktort, de *nem* teszed bele a super() hívást:**

A fordító beszúr egy super() hívást minden túlterhelt konstruktorodba*. A fordító által biztosított hívás így fest:

```
super();
```

A hívás alakja mindig ez. A fordító által beszúrt *super()* hívás mindig argumentum nélküli. Ha az alaposztály túlterhelt konstruktorokkal rendelkezik, csak az argumentum nélküli változat hívódik meg.

*Hacsak a konstruktor meg nem hív egy másik túlterhelt konstruktort (pár oldallal később ezt is láthatod).

Létezhet a gyermek a szülők előtt?

Ha az alaposztályokra úgy gondolsz, mint az alosztályaik (a gyermekeik) szülőjére, akkor kitalálhatod, hogy melyiknek kell létrejönnie előbb. ***Egy objektum alaposztályból származó részeinek teljesen fel kell épülniük, mielőtt az alosztálybeli részek létrejöhetnének.***



Ne feledd, hogy az alosztályból létrehozott objektum olyan dolgokra támaszkodhat, amelyeket az ősétől örököl, ezért fontos, hogy ezek az örökölt dolgok teljesen rendelkezésre álljanak. Ezt nem lehet megkerülni. Az alaposztály konstruktorának teljesen végeznie kell a munkájával, mielőtt az alosztály konstruktora lefuthatna. Nézd meg még egyszer a két oldallal korábbi veremábrákat. Láthatod, hogy míg a Hippo konstruktor hívódik meg *először* (ez kerül elsőként a verembe), a futása *utolsóként* fejeződik be! Minden alosztály konstruktora nyomban meghívja a saját ősosztályának a konstruktorát, amíg az Object konstruktor nem kerül a verem tetejére. Ekkor az Object konstruktor befejezi a munkáját, és visszalépünk a veremben az Animal konstruktorához. A Hippo konstruktor fennmaradó része csak az után fut le, hogy az Animal konstruktora végzett a feladatával. Ebből következik, hogy…

A super() hívásnak minden konstruktorban az első utasításnak kell lennie!*

A Boop osztály lehetséges konstruktorai

```
☑ public Boop() {
      super();
  }

☑ public Boop(int i) {
      super();
      size = i;
  }
```

Ezek rendben vannak, mert a programozó első utasításként kifejezetten megadta a super() hívást.

```
☑ public Boop() {
  }

☑ public Boop(int i) {
      size = i;
  }

⊘ public Boop(int i) {
      size = i;
      super();
  }
```

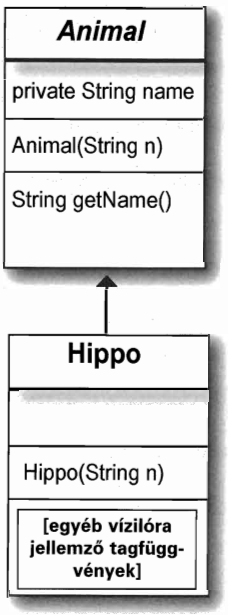
Ezek is rendben vannak, mert első utasításként a fordító beszúr egy super() hívást.

ROSSZ!!! Ez nem fordítható le! A super() hívást nem teheted semmi más alá!

*Ez alól a szabály alól van egy kivétel, amiről majd szót ejtünk.

Alaposztálybeli konstruktorok argumentumokkal

Mi történik, ha az alaposztálybeli konstruktornak argumentumai vannak? Át lehet adni valamit a *super()* hívásnak? Természetesen. Ha nem lehetne, soha nem bővíthetnél egy olyan osztályt, amelynek nincs argumentum nélküli konstruktora. Képzeld el ezt: minden állatnak van neve, és az Animal osztályban van egy *getName()* – „kiolvasNév" – nevű tagfüggvény, amely a *name* (név) példányváltozó értékét adja vissza. A példányváltozó privát, de az alosztály (esetünkben a Hippo) örökli a *getName()* tagfüggvényt. A probléma a következő: a Hippo-nak van egy (örökölt) *getName()* tagfüggvénye, de nem rendelkezik a *name* példányváltozóval. A Hippo-nak ezért önmaga Animal részére kell támaszkodnia, hogy hozzáférjen a *name* példányváltozóhoz, és visszaadhassa, ha valaki meghívja a *getName()* tagfüggvényt egy Hippo objektumra. De hogyan jut hozzá az Animal rész a névhez? A Hippo kizárólag a *super()*-en keresztül hivatkozik önmaga Animal részére, tehát ez az a hely, ahol a Hippo átadja a Hippo nevét önmaga Animal részének, hogy az Animal rész tárolhassa a privát *name* példányváltozóban.




```
public abstract class Animal {
   private String name;

   public String getName() {
      return name;
   }

   public Animal(String theName) {
      name = theName;
   }
}
```

Minden állatnak (az alosztályokat is beleértve) van neve

Kiolvasó tagfüggvény, amelyet a Hippo örököl

A konstruktor, amely fogja a nevet, és hozzárendeli azt a name példányváltozóhoz

```
public class Hippo extends Animal {

   public Hippo(String name) {
      super(name);
   }
}
```

A Hippo konstruktor egy nevet vár

a nevet felküldi a veremben az Animal konstruktorának




```
public class MakeHippo {
   public static void main(String[] args) {
      Hippo h = new Hippo("Buffy");
      System.out.println(h.getName());
   }
}
```

Itt létrehozunk egy Hippot, a „Buffy" nevet átadva a Hippo konstruktorának, majd meghívjuk a Hippo örökölt getName() tagfüggvényét

```
File Edit Window Help Hide
%java MakeHippo
Buffy
```

Túlterhelt konstruktor meghívása egy másik túlterhelt konstruktorból

Mi történik, ha túlterhelt konstruktoraid vannak, amelyek azon kívül, hogy eltérő típusú argumentumokkal dolgoznak, mind ugyanazt csinálják. Azt tudod, hogy nem szeretnél *ismétlődő* kódot a konstruktorokban (már csak a karbantartás nehézsége miatt sem), ezért a konstruktorkód nagyját (a super() hívást is beleértve) csak az *egyik* túlterhelt konstruktorban akarod elhelyezni, úgy, hogy bármelyik konstruktor hívódik is meg először, hívja meg az Igazi Konstruktort, hogy az befejezze a létrehozás feladatát. A megoldás egyszerű: csak annyit kell mondanod, hogy *this()*. Vagy azt, hogy *this(aString)*. Vagy azt, hogy *this(27, x)*. Más szóval, csak képzeld azt, hogy a this kulcsszó egy hivatkozás **az aktuális objektumra**. A *this()* kulcsszót csak egy konstruktoron belül használhatod, és annak a konstruktor első utasításának kell lennie! Ez viszont gondot jelent, nem igaz? Korábban azt mondtuk, hogy a super()-nek kell a konstruktor első utasításának lennie. Nos, ez annyit tesz, hogy választhatsz.

Ha egy konstruktort ugyanannak az osztálynak egy másik túlterhelt konstruktorából szeretnél meghívni,a this() utasítást használhatod.
A this() hívás csak konstruktorokon belül használható, és a konstruktor első utasításának kell lennie.
Egy konstruktorban lehet super() hívás VAGY this() kulcsszó, de mindkettő soha!

Minden konstruktorban lehet super() hívás vagy this() kulcsszó, de mindkettő soha!

```
class Mini extends Car {

    Color color;

    public Mini() {
        this(Color.Red);
    }

    public Mini(Color c) {
        super("Mini");
        color = c;
        // további előkészítés
    }

    public Mini(int size) {
        this(Color.Red);
        super(size);

    }
}
```

Az argumentum nélküli konstruktor alapértelmezett színt biztosít, és meghívja a túlterhelt Igazi Konstruktort (azt, amelyik meghívja a super()-t).

Ez az Igazi Konstruktor, amely Ténylegesen Elvégzi az objektum előkészítését (beleértve a super() meghívását).

Nem fog működni!!! Egy konstruktoron belül nem lehet super() és this() is, mert mindkettő magának követeli az első utasítás helyét!

```
File Edit Window Help Drive
javac Mini.java

Mini.java:16: call to super must
be first statement in constructor
        super(size);
        ^
```

Hegyezd ki a ceruzád!

A SonOfBoo osztály egyes konstruktorait nem sikerül lefordítani. Próbáld kitalálni, hogy mely konstruktorok nem érvényesek. Párosítsd össze a fordítótól kapott hibaüzeneteket a SonOfBoo-nak azokkal a konstruktoraival, amelyek előidézték őket (kösd össze egy vonallal a hibaüzenetet és a „hibás" konstruktort).

```
public class Boo {
   public Boo(int i) { }
   public Boo(String s) { }
   public Boo(String s, int i) { }
}
```

```
class SonOfBoo extends Boo {

   public SonOfBoo() {
      super("boo");
   }

   public SonOfBoo(int i) {
      super("Fred");
   }

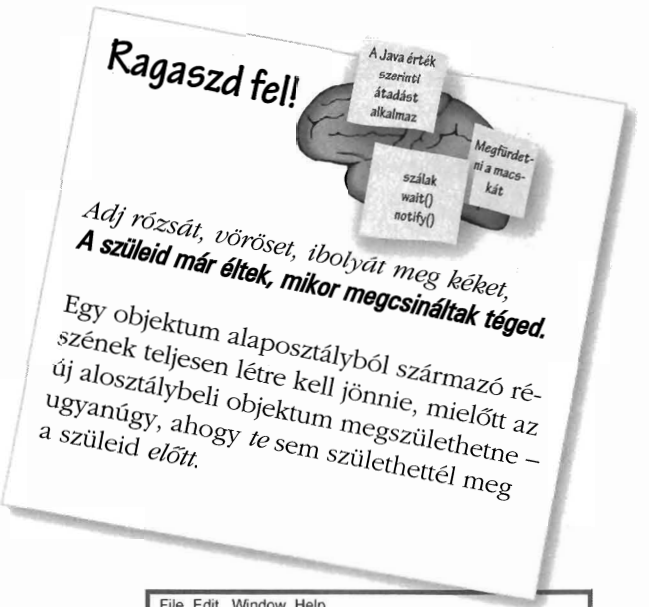
   public SonOfBoo(String s) {
      super(42);
   }

   public SonOfBoo(int i, String s) {
   }

   public SonOfBoo(String a, String b, String c) {
      super(a,b);
   }

   public SonOfBoo(int i, int j) {
      super("man", j);
   }

   public SonOfBoo(int i, int x, int y) {
      super(i, "star");
   }
}
```




```
File Edit Window Help
%javac SonOfBoo.java
cannot resolve symbol
symbol : constructor Boo
(java.lang.String,java.la
ng.String)
```

```
File Edit Window Help Yadayadayada
%javac SonOfBoo.java
cannot resolve symbol
symbol  : constructor Boo
(int,java.lang.String)
```

```
File Edit Window Help ImNotListening
%javac SonOfBoo.java
cannot resolve symbol
symbol:constructor Boo()
```

Most már tudom, hogyan születik egy objektum – de meddig él?

Egy objektum *élete* teljes mértékben a rá mutató hivatkozásoktól függ. Ha a hivatkozások „élőnek" minősülnek, az objektum is tovább él a Kupacban. Ha minden hivatkozás meghalt (azt, hogy ez mit jelent, hamarosan megtudod), az objektum is távozik az elők sorából.

Oké. Egy objektum élete a rá hivatkozó változók életétől függ – de meddig él egy *változó*?

Ez attól függ, hogy *helyi* vagy *példány*változóról van szó. Az alábbi kód egy helyi változó életét mutatja. A példában a változó primitív típusú, de a változó élettartama ugyanaz, akár primitív, akár hivatkozó változóról van szó.

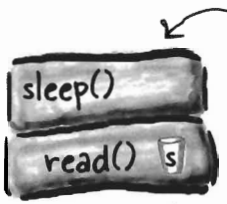
```
public class TestLifeOne {

   public void read() {
      int s = 42;
      sleep();
   }

   public void sleep() {
      s = 7;
   }
}
```

Az „s" hatóköre a read() tagfüggvény, vagyis sehol máshol nem használható

ROSSZ!!! Az „s" itt nem használható!



A sleep() nem látja az „s" változót. Mivel nem a sleep() saját veremkeretében található, a tagfüggvény semmit sem tud róla.

Az „s" változó él, de csak a read() tagfüggvény hatókörén belül. Amikor a sleep() befejezi a munkáját, és a read() a Verem tetejére kerülve folytatja a futását, a read() továbbra is látja az „s"-t. Ha viszont a read() is befejeződik, és kikerül a Veremből, az „s" meghal. Alulról szagolja a digitális ibolyát.

1 Egy helyi változó csak azon a tagfüggvényen belül él, amelyikben bevezették.

```
public void read() {
   int s = 42;
   ...
// az "s" csak ezen a tagfüggvényen
// belül használható
// Amikor a tagfüggvény befejeződik,
// az "s" teljesen eltűnik.
}
```

Az „**s**" változó *kizárólag* a *read()* tagfüggvényen belül használható. Más szavakkal, a ***változó csak a saját tagfüggvénye hatókörén belül létezik***. Az osztály (vagy bármely más osztály) semmilyen más kódja nem láthatja az „**s**"-t.

2 Egy példányváltozó addig él, amíg az objektuma. Ha az objektum még életben van, akkor a példányváltozói is.

```
public class Life {
   int size;

   public void setSize(int s) {
      size = s;
   // az "s" eltűnik, amikor
   // a tagfüggvény véget ér,
   // de a "size" az osztályban
   // bárhol használható
   }
}
```

Az „**s**" változó (amely ezúttal egy tagfüggvény paramétere) csak a setSize() tagfüggvény hatókörén belül létezik, a size példányváltozó azonban nem a *tagfüggvény*, hanem az *objektum* élete végéig marad hatókörben.

A helyi változók élete *és* hatóköre *között a következő a különbség:*

Élet

Egy helyi változó addig *él*, amíg a veremkerete megtalálható a Veremben, vagyis *amíg a tagfüggvénye be nem fejeződik.*

Hatókör

Egy helyi változó csak annak a tagfüggvénynek a *hatókörében* létezik, amelyikben bevezették. Ha a tagfüggvény egy másik tagfüggvényt hív meg, a változó életben marad, de a hatókörből eltűnik, amíg a saját tagfüggvénye nem folytatódik. ***Egy változót csak akkor használhatsz, ha hatókörben van.***

```
public void doStuff() {
   boolean b = true;
   go(4);
}

public void go(int x) {
   int z = x + 24;
   crazy();
   // képzelj ide további kódot
}

public void crazy() {
   char c = 'a';
}
```



1. **A doStuff() a Verembe kerül, tehát a „b" változó él, és hatókörben van.**



2. **A go() kerül a Verem tetejére. Az „x" és „z" változók tehát élnek, és hatókörben vannak, a „b" azonban csak él, de *nincs* hatókörben.**



3. **A Verem tetején most a *crazy()* ül, így a „c" változó él, és hatókörben van. A másik három változó is él, de a hatókörön kívül.**



4. **A *crazy()* befejeződik, és lekerül a Verem tetejéről, így a „c" kikerül a hatókörből, és *meghal*. Amikor a go() folytatja, ahol abbahagyta, az addig is élő „x" és „z" változók a hatókörbe is visszakerülnek. A „b" változó továbbra is életben van, de hatókörön kívül (amíg a go() be nem fejeződik).**

Amíg egy helyi változó él, az állapota megőrződik. Például amíg a doStuff() tagfüggvény megtalálható a Veremben, a „b" változó megtartja az értékét, de csak addig használható, amíg a doStuff() veremkerete van a verem tetején. Más szavakkal, egy helyi változót *csak* akkor használhatsz, ha a tagfüggvénye fut (és nem a felette levő veremkeretek befejezésére várakozik).

Mi a helyzet a hivatkozó változókkal?

A primitívekre és a hivatkozásokra ugyanazok a szabályok vonatkoznak. Egy hivatkozó változó csak akkor használható, ha hatókörben van, ami azt jelenti, hogy nem használhatod egy objektum távirányítóját, hacsak nincs egy hivatkozó változód a hatókörben. A *valódi* kérdés ez:

„Hogyan hat egy *változó* élete az *objektum* életére?

Egy objektum addig él, amíg vannak rá mutató élő hivatkozások. Ha egy hivatkozó változó kikerül a hatókörből, de még életben van, az objektum, amelyre *hivatkozik*, továbbra is él a Kupacban. Akkor tedd fel a kérdést: „Mi történik, ha a hivatkozást tartalmazó veremkeret lekerül a Verem tetejéről a tagfüggvény végén?"

Ha ez az *egyetlen* élő hivatkozás az objektumra, az objektum magára marad a Kupacban. A hivatkozó változó szertefoszlott a veremkerettel együtt, így a magára hagyott objektumnak *hivatalosan* is annyi. A trükk az, hogy tudd, mikor ***jelölődik ki egy objektum szemétgyűjtésre*** (garbage collection, gc).

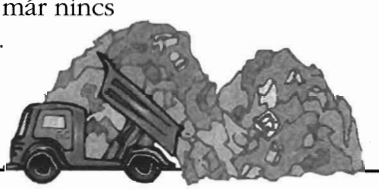
Ha egy objektum már kijelölődött szemétgyűjtésre, nem kell foglalkoznod azzal, hogy visszaköveteld az objektum által elfoglalt memóriát.

Ha a program kezd kifogyni a memóriából, a szemétgyűjtő megsemmisít néhány kijelölt objektumot (vagy az összeset), hogy megakadályozza, hogy elfogyjon a RAM. Ekkor is kifogyhatsz a memóriából, de *nem* az előtt, hogy az összes szemétgyűjtésre kijelölt objektum a szeméttelepre került volna. A te feladatod az, hogy gondoskodj az objektumok magára hagyásáról (vagyis szemétgyűjtésre kijelölhetővé tedd őket), ha már nincs rájuk szükséged, hogy a szemétgyűjtő helyet szabadíthasson fel.

Ha megőrzöd az objektumokat, a GC nem tud segíteni neked, és azt kockáztatod, hogy a programod fájdalmas halált hal, amikor megfullad a memória hiánya miatt.

Egy objektum életének nincs értéke, nincs értelme, és nincs célja, hacsak valaki nem hivatkozik rá. Ha nem tudod elérni, nem tudod megkérni sem, hogy csináljon valamit, így az objektum csak pazarolja a biteket.

Ha viszont egy objektum elérhetetlen, akkor ezt a Szemétgyűjtő észreveszi, és előbb-utóbb végez vele.



Egy objektum akkor jelölődik ki szemétgyűjtésre, amikor az utolsó élő hivatkozása is eltűnik.

Egy objektumhivatkozás három módon semmisülhet meg:

1. A hivatkozás véglegesen kikerül a hatókörből:

```
void go() {
  Life z = new Life();
}
```

a „z" hivatkozás meghal a tagfüggvény végén

2. A hivatkozást egy másik objektumhoz rendeled:

```
Life z = new Life();
z = new Life();
```

az első objektum magára marad, amikor a „z"-t „átprogramozod" egy másik objektumra

3. A hivatkozást kifejezetten null-ra állítod:

```
Life z = new Life();
z = null;
```

az első objektum magára marad, amikor a „z" programozását „kikapcsolod"

Objektumgyilkos #1

A hivatkozás véglegesen kikerül a hatókörből



```
public class StackRef  {
   public void foof() {
      barf();
   }

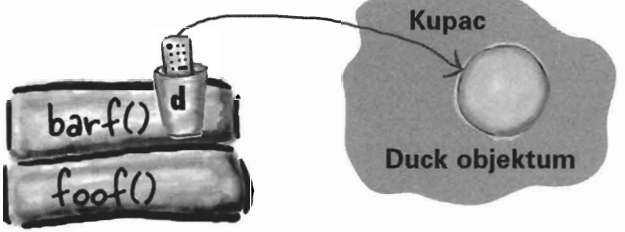
   public void barf() {
      Duck d = new Duck();
   }
}
```



1 A Verem tetejére a *foof()* kerül, bejelentett változók nincsenek.

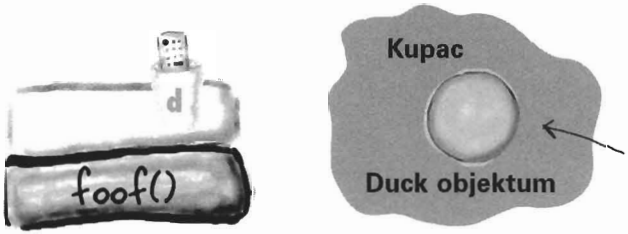


2 A Verem tetejére a *barf()* kerül, ahol bevezet egy hivatkozó változót, és létrehoz egy ahhoz hozzárendelt új objektumot. Az objektum a Kupacban jön létre, a hivatkozás pedig él, és hatókörben van.



Az új Duck objektum a Kupacba kerül, és amíg a barf() fut, a „d" hivatkozás él, és hatókörben marad, így a Kacsa is élőnek minősül.

3 A *barf()* befejezi a működését, és kikerül a Veremből. A kerete szertefoszlik, így a „d" változó is megsemmisül. A végrehajtás visszatér a *foof()*-hoz, de az nem használhatja „d"-t.



Ó, jaj. A „d" változó eltávozott, amikor a barf() veremkerete lekerült a veremről, így a Kacsa magára maradt, kiszolgáltatva a szemétgyűjtőnek.

Objektumgyilkos #2

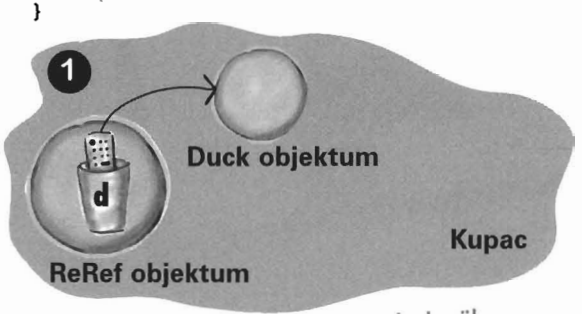
A hivatkozás másik objektumhoz rendelése



```
public class ReRef {

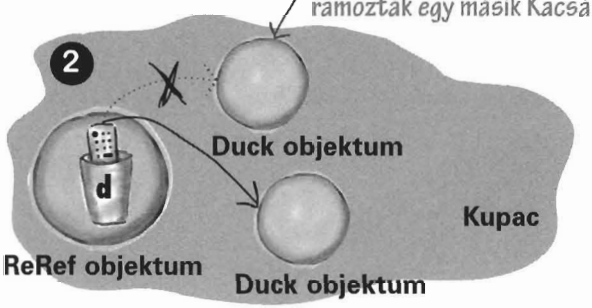
    Duck d = new Duck();

    public void go() {
      d = new Duck();
    }
}
```



A „d" által hivatkozott új Kacsa a Kupacba kerül. Mivel „d" példányváltozó, a Kacsa addig él, amíg az őt példányosító ReRef objektum is életben van. Hacsak...

Amikor valaki meghívja a go() tagfüggvényt, ez a Kacsa magára marad. Az egyetlen rá mutató hivatkozást átprogramozták egy másik Kacsára.



A „d" változót egy másik Duck objektumhoz rendelték, így az eredeti (az első) Kacsa magára maradt. Az első Kacsának tehát lényegében lőttek.



Objektumgyilkos #3

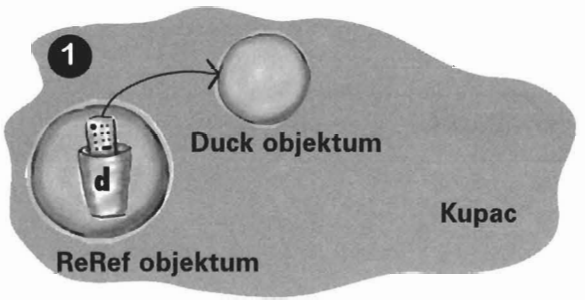
A hivatkozás kifejezett null-ra állítása



```
public class ReRef {

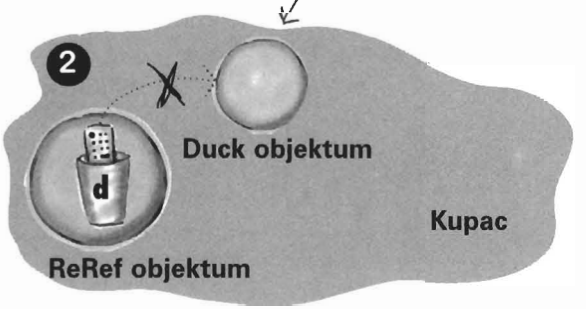
    Duck d = new Duck();

    public void go() {
      d = null;
    }
}
```



A „d" által hivatkozott új Kacsa a Kupacba kerül. Mivel „d" példányváltozó, a Kacsa addig él, amíg az őt példányosító ReRef objektum is életben van. Hacsak...

Ez a Kacsa magára maradt. Az egyetlen rá mutató hivatkozást null-ra állították.



A „d" változót null-ra állították, ami olyan, mintha a távirányító nem lenne beprogramozva semmire. Még a pont műveletet sem használhatod a „d"-re, amíg át nem programozod (vagyis hozzá nem rendeled egy másik objektumhoz).

A `null` jelentése

Amikor egy hivatkozást **`null`** értékre állítasz, kikapcsolod a távirányító programozását. Más szavakkal, lesz egy távirányítód, amihez viszont nem tartozik TV. A null vagy üres hivatkozás olyan biteket tartalmaz, amelyek a „semmit" jelképezik (*mi* nem tudjuk, milyen bitek is ezek, és nem is érdekel minket – a lényeg, hogy a JVM tudja).

A való világban egy be nem programozott távirányító gombjai nem csinálnak semmit, ha nyomkodod őket. A Javában azonban nem is nyomhatod meg a gombokat (vagyis nem használhatod a pont műveletet egy üres hivatkozásra), mert a JVM tudja (futásidőben – ez nem a fordítóra tartozó kérdés), hogy ugatást várs, de nincs Kutya, ami ugathatna.

Ha a pont műveletet egy üres hivatkozásra alkalmazod, futtatáskor NullPointerException kivételt kapsz.

A kivételekről a 11. fejezetben (*Kockázatos viselkedés*) lesz szó.

Kandalló melletti csevely





Ma esti vendégeink, **egy példányváltozó és egy helyi változó, az élet és a halál kérdéséről vitatkoznak (meglepően civilizáltan)**

Példányváltozó

Én szeretnék megszólalni először, mert általában fontosabb vagyok egy program számára, mint egy helyi változó. A feladatom az, hogy támogassak egy objektumot, általában az objektum egész életén keresztül. Végtére is, mi egy objektum *állapot* nélkül? És mi az állapot? ***Példányváltozókban*** tárolt értékek.

Helyi változó

Tiszteletben tartom az Ön nézőpontját, és természetesen nagyra értékelem az objektumok állapotjellemzőit, meg minden, de nem szeretném, ha félrevezetnénk a nézőket. A helyi változók *igenis* fontosak. Hogy az Ön szófordulatával éljek: „Végtére is, mi egy objektum *viselkedés* nélkül?" És mi a viselkedés? Tagfüggvényekben elhelyezett algoritmusok. Arra pedig mérget vehet, hogy ahhoz, hogy ezek az algoritmusok működjenek, szükség van néhány *helyi változóra.*

Ön félreértett. Én értem az Ön szerepét a tagfüggvényekben, csak arra céloztam, hogy az Ön élete olyan rövid, olyan...átmeneti. Ezért is hívják Önt sokszor „ideiglenes változónak".

A helyi változók közösségében az „ideiglenes változó" kifejezés sértőnek számít. Jobban szeretjük a „helyi", „verem-" „automatikus" vagy „hatóköri kihívásokkal küszködő" megnevezést.

Elnézését kérem. Tökéletesen megértem.

Az igaz, hogy az életünk nem hosszú, és nem is különösebben *boldog.* Először is, belapátolnak minket egy veremkeretbe az összes többi helyi változóval együtt. Majd, ha a tagfüggvény, amelynek a részei vagyunk, meghív egy másik tagfüggvényt, egy újabb keretet pakolnak a fejünkre. És ha *az* a tagfüggvény is meghív egy *másik* tagfüggvényt...Néha egy örökkévalóságig tart kivárni, amíg a Verem tetején levő összes tagfüggvény végez, hogy a mi tagfüggvényünk folytathassa a munkáját.

Ezt még soha nem gondoltam így végig. Mit csinál, miközben a többi tagfüggvény fut, és Ön arra vár, hogy a kerete ismét a Verem tetején legyen?

Semmit. Olyan, mintha hibernáltak volna – tudja, mint a sci-fi filmekben, amikor valaki nagy távolságra utazik. Lényegében kómában vagyok, és csak fekszem mozdulatlanul. Amíg a keretem létezik, biztonságban vagyok, akárcsak a bennem tárolt érték, de amikor a keretem újra futhat, az vegyes érzéseket kelt bennem.

Példányváltozó

Egyszer láttam róla egy oktatófilmet. Elég brutális végnek tűnt. Úgy értem, amikor a tagfüggvény elér a záró kapcsos zárójeléhez, a keret szó szerint *lerobban* a Veremről! Biztosan fájdalmas.

A Kupacban, az objektumokkal. Pontosabban nem *velük*, hanem *bennük*, egy objektumban. Abban az objektumban, amelyiknek az állapotát tárolom. Be kell vallanom, hogy az élet elég fényűző tud lenni a Kupacban. Sokan közülünk bűntudatot is éreznek, különösen az ünnepek idején.

Jó, tegyük fel – persze csak elméletben –, hogy a Nyakörv példányváltozója vagyok, és a Nyakörv kijelölődik szemétgyűjtésre. Ekkor a Nyakörv példányváltozóit valóban úgy dobják ki, mint egy pizzásdobozt. De nekem azt mondták, hogy erre szinte soha nem kerül sor.

Nekünk szabad *innunk*?

Helyi változó

Egyrészt jó, hogy ismét aktív lehetek, másrészt azonban rövid életem órája ismét ketyegni kezd. Minél tovább fut a tagfüggvényem, annál közelebb kerül a végéhez, és *mindnyájan* tudjuk, mi történik akkor.

Nekem mondja? A számítógép-tudományban a *pop* kifejezést használják, mintha a keretet úgy pattintanák le („pop„) a Veremről, mint egy kupakot. Szórakoztatónak hangzik – egyfajta extrém sportnak. Nos, Ön viszont látta a filmet. De miért nem beszélünk magáról? A saját keretemet ismerem, de hol él *Ön*?

De nem *mindig* él ugyanaddig, mint az Önt bevezető objektum, ugye? Tegyük fel például, hogy van egy Kutya objektum egy Nyakörv példányváltozóval, *Ön* pedig a *Nyakörv* objektum egy példányváltozója, mondjuk egy hivatkozás egy Csatra vagy hasonlóra. Boldogan él a *Nyakörv* objektumban, az pedig a *Kutya* objektumban. Csakhogy…mi történik, ha a Kutya új Nyakörvet akar, vagy *null*-ra állítja a Nyakörv példányváltozóját? Ekkor a Nyakörv objektum kijelölődik szemétgyűjtésre. Ha tehát *Ön* egy példányváltozó a Nyakörvön belül, és az egész *Nyakörvet* eldobták, mi lesz az *Ön* sorsa?

És maga elhitte? Ezt csak azért mondják, hogy motiváltak és hatékonyak maradjunk. De nem feledkezett meg valamiről? Mi a helyzet akkor, ha Ön egy példányváltozó egy objektum belsejében, amelyre *csak* egy *helyi* változó hivatkozik? Ha én vagyok az egyetlen hivatkozás arra az objektumra, amelyikben Ön található, akkor az én halálommal az Ön élete is véget ér. Tetszik, nem tetszik, megeshet, hogy a sorsunk össze van kötve. Ezért azt mondom, feledkezzünk meg erről az egészről, és rúgjunk be, amíg lehet. „Carpe RAM", vagy ilyesmi.



Légy a Szemétgyűjtő!



A bal oldali osztályba az A ponton beszúrva mely jobb oldali kódsorok jelölnének ki pontosan egy további objektumot szemétgyűjtésre? (Feltételezve, hogy az A pont – // további tagfüggvények meghívása – végrehajtása sokáig tart, ami időt biztosít a Szemétgyűjtőnek, hogy elvégezze a feladatát.)

```
public class GC {
  public static GC doStuff() {
    GC newGC = new GC();
    doStuff2(newGC);
    return newGC;
  }

  public static void main(String [] args) {
    GC gc1;
    GC gc2 = new GC();
    GC gc3 = new GC();
    GC gc4 = gc3;
    gc1 = doStuff();

    A

    // további tagfüggvények meghívása
  }

  public static void doStuff2(GC copyGC) {
    GC localGC
  }
}
```

1 `copyGC = null;`

2 `gc2 = null;`

3 `newGC = gc3;`

4 `gc1 = null;`

5 `newGC = null;`

6 `gc4 = null;`

7 `gc3 = gc2;`

8 `gc1 = gc4;`

9 `gc3 = null;`



Gyakorlat

Népszerű objektumok

Ebben a kódpéldában néhány új objektum jön létre. A feladatod az, hogy megkeresd a „legnépszerűbb" objektumot, vagyis azt, amelyikre a legtöbb hivatkozó változó mutat, majd felsorold, hogy pontosan mely hivatkozások mutatnak az objektumra, és összesen *hány* van belőlük. Segítségképpen rámutattunk az egyik új objektumra és a rá hivatkozó változóra. Sok szerencsét!

```
class Bees {
  Honey [] beeHA;
}

class Raccoon {
  Kit k;
  Honey rh;
}

class Kit {
  Honey kh;
}

class Bear {
  Honey hunny;
}

public class Honey {
  public static void main(String [] args) {
    Honey honeyPot = new Honey();
    Honey [] ha = {honeyPot, honeyPot, honeyPot, honeyPot};
    Bees b1 = new Bees();
    b1.beeHA = ha;
    Bear [] ba = new Bear[5];
    for (int x=0; x < 5; x++) {
      ba[x] = new Bear();
      ba[x].hunny = honeyPot;
    }
    Kit k = new Kit();
    k.kh = honeyPot;
    Raccoon r = new Raccoon();

    r.rh = honeyPot;
    r.k = k;
    k = null;
  }   // a main vége
}
```

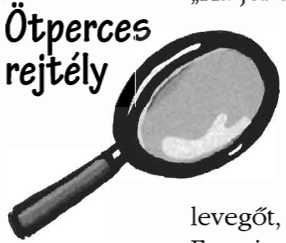
Ez egy új Raccoon objektum!

Ez pedig a rá hivatkozó „r" változó.



„Négyszer futtattuk le a szimulációt, és a főmodul hőmérséklete mindig eltért a hideg felé." – mondta Sarah kimerülve. – „A múlt héten telepítettük az új hőmérséklet-szabályozó robotokat. A radiátorbotok, amelyeket a lakómodul hűtésére terveztek, úgy tűnik, az előírt értékeket mutatják, ezért az elemzés során a hőtartó robotokra összpontosítottunk, amelyek a modul fűtését segítik." Tom felsóhajtott. Végre úgy tűnt, a nanotechnológia tényleg segít behozni a lemaradást. Már csak öt hét maradt a kilövésig, és a keringőegység egyes kulcsfontosságú életfenntartó rendszerei még mindig nem teljesítették a szimulációs teszteket.

„Milyen aránnyal végeztétek a szimulációt?" – kérdezte Tom.



„Ha jól értem, amire célzol, már foglalkoztunk a kérdéssel." – válaszolta Sarah. – „A küldetésirányítók nem fogják rányomni a pecsétet a létfontosságú rendszerekre, ha nem az előírt kereteken belül működtetjük őket. A v3 radiátorbot szimulátoregységeit 2:1 arányban kell futtatni a v2 radiátor szimulátoregységeihez képest. Összesítve, a hőtartó robotok aránya a radiátorbotokhoz képest 4:3 kell legyen."

„Hogy áll az energiafogyasztás, Sarah?" – kérdezte Tom. Sarah vett egy mély levegőt, aztán kibökte: „Na, az a másik. Az energiafogyasztás magasabb a vártnál. Ezen is dolgozik egy csapat, hogy kiderítsék az okát, de mivel a nanók drót nélküliek, nehéz elkülöníteni a radiátorok energiafogyasztását a hőtartó robotokétól. Az összesített energiafogyasztási arány" – folytatta Sarah – „a terv szerint 3:2 kell legyen, és a radiátoroknak kell több energiát leszívniuk a drót nélküli rácsból."

„Oké, Sarah" – bólintott Tom – „nézzük meg a szimuláció előkészítő kódját. Meg kell találnunk, mi okozza a gondot, méghozzá gyorsan!"

```
import java.util.*;
class V2Radiator {
  V2Radiator(ArrayList list) {
    for(int x=0; x<5; x++) {
      list.add(new SimUnit("V2Radiator"));
    }
  }
}

class V3Radiator extends V2Radiator {
  V3Radiator(ArrayList lglist) {
    super(lglist);
    for(int g=0; g<10; g++) {
      lglist.add(new SimUnit("V3Radiator"));
    }
  }
}

class RetentionBot {
  RetentionBot(ArrayList rlist) {
    rlist.add(new SimUnit("Retention"));
  }
}
```

Ötperces rejtély folytatás...

```
public class TestLifeSupportSim {
  public static void main(String [] args) {
    ArrayList aList = new ArrayList();
    V2Radiator v2 = new V2Radiator(aList);
    V3Radiator v3 = new V3Radiator(aList);
    for(int z=0; z<20; z++) {
      RetentionBot ret = new RetentionBot(aList);
    }
  }
}

class SimUnit {
  String botType;
  SimUnit(String type) {
    botType = type;
  }
  int powerUse() {
    if ("Retention".equals(botType)) {
      return 2;
    } else {
      return 4;
    }
  }
}
```

Ahogy Tom átfutotta a kódot, mosolyra húzódott a szája. „Azt hiszem, megvan a probléma, Sarah. És lefogadom, hogy azt is meg tudom mondani, mekkora százalékkal tér el a mért energiafogyasztás!"

Mire gyanakodott Tom? Honnan találhatta ki, mi a hiba a leolvasott energiafogyasztási értékekben? Milyen kódsorok hozzáadásával segíthetnél kiküszöbölni a hibát a programban?

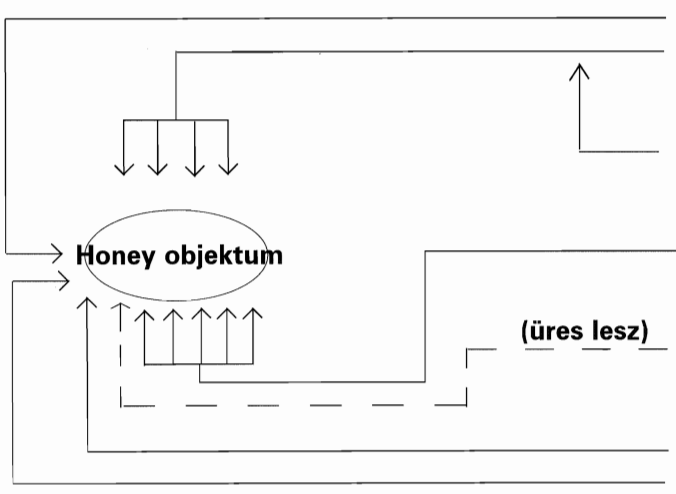


Gyakorlat – Megoldás

1	`copyGC = null;`	Nem – ez a sor egy hatókörön kívüli változót próbál elérni.
2	`gc2 = null;`	OK – erre az objektumra csak a gc2 változó hivatkozik.
3	`newGC = gc3;`	Nem – ez is hatókörön kívüli változó.
4	`gc1 = null;`	OK – a gc1 az egyetlen hivatkozás, mivel a newGC hatókörön kívül van.
5	`newGC = null;`	Nem – a newGC hatókörön kívül van.
6	`gc4 = null;`	Nem – a gc3 továbbra is hivatkozik az objektumra.
7	`gc3 = gc2;`	Nem – a gc4 továbbra is hivatkozik az objektumra.
8	`gc1 = gc4;`	OK – máshoz rendeli az objektum egyetlen hivatkozását.
9	`gc3 = null;`	Nem – a gc4 továbbra is hivatkozik az objektumra.

Népszerű objektumok

Valószínűleg nem volt túl nehéz kitalálni, hogy a Honey objektum, amelyre először a honeyPot változó hivatkozik, messze a „legnépszerűbb" objektum ebben az osztályban. Az viszont talán nem volt ilyen nyilvánvaló, hogy a kódból a Honey objektumra mutató összes változó ***ugyanarra az objektumra*** hivatkozik! Összesen 12 aktív hivatkozás mutat erre az objektumra, mielőtt a main() tagfüggvény befejeződne. A *k.kh* változó egy darabig érvényes, de a *k* a végén lenullázódik. Mivel az *r.k* továbbra is a Kit objektumra mutat, az *r.h.kh* (bár kifejezetten soha nem lett így bevezetve) is az objektumra hivatkozik!




```
public class Honey {
  public static void main(String [] args) {
    Honey honeyPot = new Honey();
    Honey [] ha = {honeyPot, honeyPot,
                   honeyPot, honeyPot};
    Bees b1 = new Bees();
    b1.beeHA = ha;
    Bear [] ba = new Bear[5];
    for (int x=0; x < 5; x++) {
      ba[x] = new Bear();
      ba[x].hunny = honeyPot;
    }
    Kit k = new Kit();
    k.kh = honeyPot;
    Raccoon r = new Raccoon();

    r.rh = honeyPot;
    r.k = k;
    k = null;
    // a main vége
  } }
```

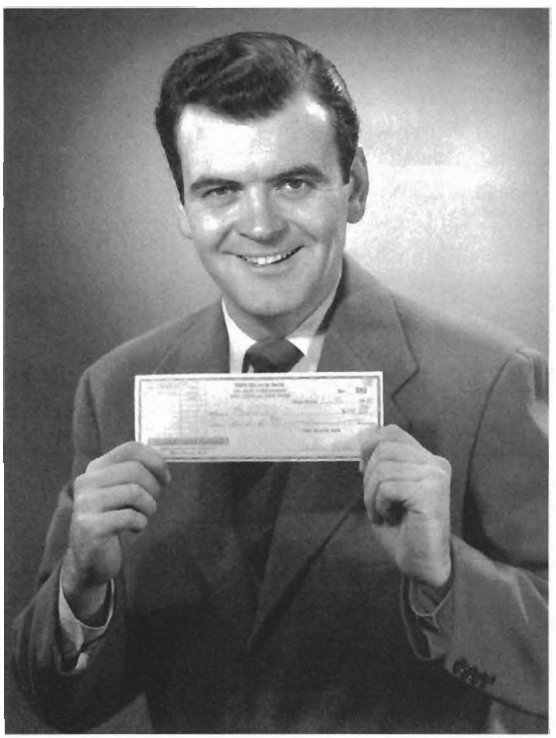


Az Ötperces rejtély megoldása

Tom észrevette, hogy a V2Radiator osztály konstruktora egy tömblistát (ArrayList) vár. Ez azt jelenti, hogy a *V3*Radiator konstruktora minden meghívásakor egy tömblistát ad át a super() hívásban a *V2*Radiator konstruktorának, ami további öt V2Radiator szimulátoregység (SimUnit) létrehozását eredményezi. Ha Tomnak igaza van, akkor a teljes energiafogyasztás 120, és nem 100, mint ahogy Sarah elvárt arányai jósolták.

Mivel minden Bot-osztály szimulátoregységeket hoz létre, a problémára gyorsan rá lehetett volna világítani, ha a SimUnit osztály számára olyan konstruktort írnak, amely minden alkalommal kiír egy sornyi szöveget, amikor létrejön egy SimUnit!

A számok számítanak



Számolj! A számokkal való munka persze többet jelent a primitív számtani műveletek elvégzésénél. Előfordulhat, hogy egy szám abszolút értékét szeretnéd megkapni, kerekítést szeretnél végezni, vagy meg szeretnéd határozni, hogy két szám közül melyik a nagyobb. Esetleg azt szeretnéd, hogy bizonyos számok pontosan két tizedesjeggyel íródjanak ki, vagy a könnyebb olvashatóság érdekében tagolni akarod a nagy számokat. És mi a helyzet a dátumokkal? Az is megeshet, hogy a dátumokat többféleképpen szeretnéd megjeleníteni, sőt akár *műveleteket* is szeretnél végezni rajtuk, például három hetet hozzáadni a mai dátumhoz. És a karakterláncok számmá – vagy éppen egy szám karakterlánccá – alakítása? Nos, szerencséd van: a Java API telis-tele van előre elkészített, könnyen használható számbűvölő tagfüggvényekkel. A legtöbbjük azonban **statikus**, ezért először azt tisztázzuk, hogy mit jelent az, ha egy változó vagy tagfüggvény statikus, és mik azok az állandók (konstansok): statikus *végleges* változók.

A Math osztály tagfüggvényei: a leginkább globális tagfüggvények

A Javában persze *semmi* sem globális, de gondolj rájuk így: olyan tagfüggvények, amelyeknek a viselkedése nem függ egy példányváltozó értékétől. Vegyük például a Math osztály round() tagfüggvényét. Ez a tagfüggvény minden alkalommal ugyanazt csinálja: a legközelebbi egészre kerekít egy lebegőpontos számot (a tagfüggvény argumentumát). Mindig. Ha 10 000 példányod van a Math osztályból, és mindegyikben lefuttatod a round(42.2) utasítást, minden alkalommal a 42 egész értéket kapod. Más szavakkal, a tagfüggvény az argumentumán végez műveletet, de soha nem hat rá egy példányváltozó állapota. Az egyetlen érték, ami megváltoztatja a round() tagfüggvény lefutását, a függvénynek átadott argumentum.

Nem tűnik hasznos kupacterület pazarlásának példányt létrehozni a Math osztályból csak azért, hogy lefuttassuk a round() tagfüggvényt? És mi a helyzet a Math *más* tagfüggvényeivel, például a min()-nel, amely két primitív szám közül a kisebbet adja vissza? És a max()-szal? Vagy a szám abszolút értékét visszaadó abs() tagfüggvénnyel?

Ezek a tagfüggvények soha nem használják példányváltozók értékét.

Valójában a Math osztálynak egyáltalán *nincsenek* is példányváltozói. Azzal tehát semmit sem nyerünk, ha példányt hozunk létre a Math osztályból. De tudod mit? Nem is kell – sőt nem is lehet.

A Math osztályban található tagfüggvények nem használnak példányváltozó-értékeket. Mivel ezek a tagfüggvények „statikusak", a Math osztályból nem kell példányt létrehoznod. Csak a Math osztályra van szükséged.

```
int x = Math.round(42.2);
int y = Math.min(56,12);
int z = Math.abs(-343);
```

Ezek a tagfüggvények soha nem használnak példányváltozókat, ezért a műveleteikhez nincs szükség konkrét objektumra.

Ha példányt próbálsz létrehozni a Math osztályból:

```
Math mathObject = new Math();
```

Ezt a hibaüzenetet kapod:

```
File Edit Window Help IwasToldThereWouldBeNoMath
%javac TestMath

TestMath.java:3: Math() has private
access in java.lang.Math

     Math mathObject = new Math();
                       ^

1 error
```

Ez a hibaüzenet azt mutatja, hogy a Math konstruktora privát megjelölésű! Ez azt jelenti, hogy SOHA nem mondhatsz „new"-t a Math osztályra, hogy létrehozz belőle egy új Math objektumot.

Mi a különbség a normál (nem statikus) és a statikus tagfüggvények között?

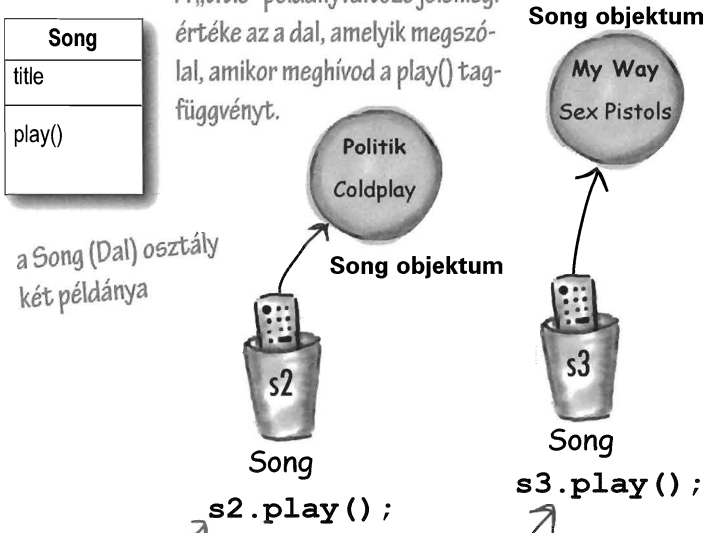
A Java objektumközpontú, de itt-ott különleges elemekkel is találkozhatsz – jellemzően segédfüggvényekkel (amilyenek a Math osztály tagfüggvényei is) –, amelyeknek nincs szükségük az osztály egy példányára.
A **static** kulcsszó lehetővé teszi, hogy egy tagfüggvényt ***osztálypéldány nélkül*** futtass. Ha egy tagfüggvény statikus, az azt jelenti, hogy „a tagfüggvény viselkedése nem függ példányváltozóktól, ezért a futtatásához nincs szükség példányra/objektumra, csupán az osztályra".

normál (nem statikus) tagfüggvény

```
public class Song {
   String title;
   public Song(String t) {
     title = t;
   }
   public void play() {
      SoundPlayer player = new SoundPlayer();
      player.playSound(title);
   }
```

A példányváltozók értéke befolyásolja a play() tagfüggvény viselkedését.

A „title" példányváltozó jelenlegi értéke az a dal, amelyik megszólal, amikor meghívod a play() tagfüggvényt.



a Song (Dal) osztály két példánya

```
s2.play();
```

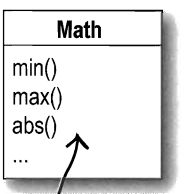
Ha a play()-t erre a hivatkozásra hívod meg, a „Politik" csendül fel.

```
s3.play();
```

Ha a play()-t erre a hivatkozásra hívod meg, a „My Way" szólal meg.

statikus tagfüggvény

```
public static int min(int a, int b){
// a kisebbet adja vissza a és b közül
}
```



Nincsenek példányváltozók. A tagfüggvény viselkedése nem változik példányváltozók állapotától függően.

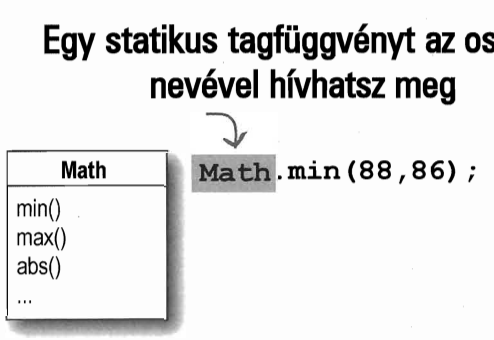
```
Math.min(42,36);
```

Az osztály, és nem egy hivatkozó változó nevét kell használnod.



NINCSENEK OBJEKTUMOK!!!

Ezen a képen egyáltalán NINCSENEK OBJEKTUMOK!



Mit jelent az, ha egy osztálynak statikus tagfüggvényei vannak?

A statikus tagfüggvényekkel rendelkező osztályokat gyakran (bár nem mindig) nem arra szánták, hogy példányokat hozz létre belőlük. A 8. fejezetben beszéltünk az elvont osztályokról, illetve arról, hogy ha egy osztályt az **`abstract`** módosítóval jelölsz meg, akkor senki sem mondhat „new"-t az említett osztálytípusra. Más szavakkal, ***egy elvont osztályt lehetetlen példányosítani***.

Ugyanakkor azt is megakadályozhatod, hogy más kódok egy *nem* elvont osztály példányosítsanak, ha a konstruktorát **`private`** jelzéssel látod el. Emlékezz vissza, hogy ha egy *tagfüggvényt* privátként jelölsz meg, az azt jelenti, hogy csak az adott osztályban található kódok hívhatják meg. A privátként megjelölt *konstruktorokra* lényegében ugyanez vonatkozik: csak az osztály belsejében található kódok hívhatják meg; az osztályon *kívülről* senki nem mondhat „new"-t. Így működik például a Math osztály. A konstruktora privát, ezért a Math osztályból nem hozható létre új példány. A fordítóprogram tudja, hogy a kódod nem férhet hozzá ehhez a privát konstruktorhoz.

Ez *nem* azt jelenti, hogy egy osztályt, amelynek egy vagy több statikus tagfüggvénye van, soha nem szabd példányosítani. Valójában minden olyan osztály, amelyben elhelyezel egy main() tagfüggvényt, rendelkezik statikus tagfüggvénnyel!

A main() tagfüggvényeket jellemzően azért helyezed el, hogy elindíthass vagy tesztelhess egy másik osztályt, szinte mindig úgy, hogy példányosítod azt a main()-ben, majd erre az új példányra meghívsz egy tagfüggvényt. A statikus és nem statikus tagfüggvényeket tehát vegyítheted egy osztályon belül, bár akár csak egyetlen nem statikus tagfüggvény jelenléte is azt jelenti, hogy *valamilyen* módnak kell lennie arra, hogy példányt hozz létre az osztályból. Új objektumot kizárólag a „new" paranccsal vagy visszafejtéssel kaphatsz (illetve a Java Reflection API segítségével, amibe most nem megyünk bele). Nincs más út. Az azonban, hogy pontosan ki mond „new"-t, érdekes kérdés lehet – a fejezet későbbi részében ki is térünk rá.

A statikus tagfüggvények nem használhatnak nem statikus (példány-) változókat!

A statikus tagfüggvények anélkül működnek, hogy tudnának az osztályuk bármilyen konkrét példányáról – és ahogy az előző oldalon láttad, lehet, hogy *nincsenek* is ilyen példányok. Mivel a statikus tagfüggvényeket az *osztály* nevével (**Math**.random()), és nem egy *példányváltozóéval* (**t2**.play()) hívjuk meg, egy statikus tagfüggvény nem hivatkozhat az osztály egyetlen példányváltozójára sem, mivel nem tudja, hogy *melyik* példány változóértékét kellene használni.

Ha egy példányváltozót egy statikus tagfüggvényből próbálsz használni, a fordító nem fogja tudni, hogy melyik objektum példányváltozójára gondolsz. Ha van mondjuk tíz Duck objektumod a kupacban, a statikus tagfüggvények egyikről sem tudnak.

Ha megpróbálod lefordíttatni ezt a kódot:

```
public class Duck {

   private int size;

   public static void main (String[] args) {
      System.out.println("Size of duck is " + size);
   }

   public void setSize(int s) {
      size = s;
   }
   public int getSize() {
       return size;
   }
}
```

Melyik Kacsa? Melyiknek a mérete?

Ha a kupacban van is valahol egy Kacsa, nem tudunk róla.

Ezt a hibaüzenetet kapod:

```
File Edit Window Help Quack
% javac Duck.java
Duck.java:6: non-static variable size cannot be referenced from a static context
      System.out.println("Size of duck is " + size);
                                              ^
```



A statikus tagfüggvények nem használhatnak nem statikus tagfüggvényeket sem!

Hogyan működnek a nem statikus tagfüggvények? A ***viselkedésük általában módosul a példányváltozókban tárolt állapotnak megfelelően.*** Egy getName() nevű tagfüggvény például a name változó értékét adja vissza – de minek a nevét? A getName() tagfüggvény meghívására használt objektumét.

Ezt nem lehet lefordítani:

```
public class Duck {

   private int size;

   public static void main (String[] args) {
      System.out.println("Size is " + getSize());
   }

   public void setSize(int s) {
      size = s;
   }
   public int getSize() {
       return size;
   }
}
```

A getSize() meghívása csak késlelteti az elkerülhetetlent – a getSize() a size példányváltozót használja.

Ugyanaz a probléma... minek a mérete?

```
File Edit Window Help Jack-in
% javac Duck.java
Duck.java:6: non-static method
getSize() cannot be referenced
from a static context
       System.out.println("Size
of duck is " + getSize());
                    ^
```



Nincsenek hülye kérdések

K: És ha egy nem statikus tagfüggvényt egy statikus tagfüggvényből próbálok meghívni, és a nem statikus tagfüggvény nem használ semmilyen példányváltozót? Ezt megengedi a fordító?

V: Nem. A fordító tudja, hogy akár használsz példányváltozókat egy nem statikus tagfüggvényben, akár nem – *megteheted.* Gondold végig, mi következik ebből... Mi történne például, ha a fordító megengedné, amit fent vázoltál, és a jövőben módosítanád a nem statikus tagfüggvény megvalósítását, hogy *mégis* használjon egy példányváltozót? Vagy ami még rosszabb, mi történik, ha egy alosztály *felülírja* a tagfüggvényt, és a felülíró változat használ példányváltozókat?

K: Meg mernék esküdni rá, hogy már láttam olyan kódot, ami hivatkozó változóval, és nem osztálynévvel hívott meg egy statikus tagfüggvényt.

V: *Lehetséges*, de jusson eszedbe, amit anyád mondogatott: „attól, hogy szabad, még nem biztos, hogy jó". Egy statikus tagfüggvényt meg lehet hívni az osztály bármely példányára, de a kód félrevezető (és kevésbé egyértelmű) lesz. *Mondhatod* például ezt:

```
Duck d = new Duck();
String[] s = {};
d.main(s);
```

Ez a kód érvényes, de a fordító úgyis visszafordítja az igazi osztályra. („Jó, szóval a *d* Duck típusú, a main() pedig statikus, ezért a Duck osztály statikus main() tagfüggvényét fogom meghívni.") Más szavakkal, attól még, hogy a *d*-vel hívod meg a main()-t, a main() nem fogja tudni, hogy a *d* milyen objektumra hivatkozik. Ez csupán egy másik *módja* egy statikus tagfüggvény meghívásának; a tagfüggvény ettől még statikus marad!

Statikus változók:

amikor az érték az osztály MINDEN példányában ugyanaz

Tegyük fel, hogy meg szeretnéd számolni, hány Kacsa jön létre a programod futása alatt. Hogyan oldanád meg? Esetleg egy példányváltozóval a konstruktorban, aminek mindig növeled az értékét eggyel?

```
class Duck {
    int duckCount = 0;
     public Duck() {
        duckCount++;
    }
}
```

ez a duckCount-ot mindig 1-re állítja, amikor létrejön egy Duck

Ez sajnos nem működik, mert a duckCount (kacsaSzámláló) példányváltozó, így minden Duck objektum esetében 0-ról indul az értéke. Megpróbálhatod, hogy egy másik osztályból hívsz meg egy tagfüggvényt, de ez körülményes megoldás. Egy olyan osztályra van szükséged, amelyben csak egy példány található a változóból, és minden objektumpéldány ezt a változópéldányt használja.

Erre jó egy statikus változó: egy olyan értéket biztosít, amin az osztály összes példánya osztozhat. Más szavakkal, nem *példányonként*, hanem *osztályonként* kapsz egy értéket.

A statikus duckCount változó CSAK az osztály első betöltésekor kap kezdőértéket, NEM minden alkalommal, amikor létrejön egy új példány.

```
public class Duck {

    private int size;
    private static int duckCount = 0;

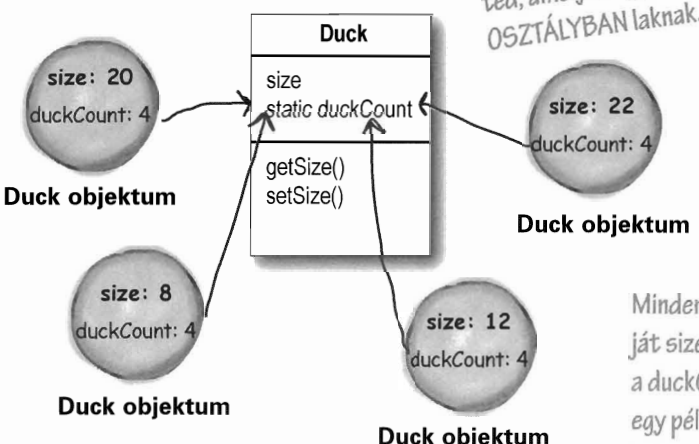
    public Duck() {
        duckCount++;
    }

    public void setSize(int s) {
        size = s;
    }
    public int getSize() {
        return size;
    }
}
```

Most már a Duck konstruktor minden lefutásakor növelni fogja az értéket, mert a duckCount statikus, így nem fog visszaállni 0-ra.

A Duck objektumok nem tartanak fenn saját példányt a duckCount-ból.

Mivel a duckCount statikus, a Duck objektumok egyetlen példányán osztoznak. A statikus változókat olyan változóknak tekintheted, amelyek egy objektum helyett egy OSZTÁLYBAN laknak.



Minden Duck objektumnak saját size változója van, a duckCount-ból azonban csak egy példány létezik – az, amelyik az osztályban található.



A statikus változókon osztoznak az objektumok.

Az azonos osztályú objektumok egyetlen példányt használnak a statikus változókból.

példányváltozók: **példányonként** 1
statikus változók: **osztályonként** 1



Agygyúró

A fejezet korábbi részében megtanultad, hogy a privát konstruktor azt jelenti, hogy az osztály nem példányosítható az osztályon kívülről. Más szavakkal, csak az osztályon belül található kódrészek hozhatnak létre új példányt egy olyan osztályból, amelynek privát a konstruktora. (Ez persze egyfajta „Melyik volt előbb: a tyúk vagy a tojás?" probléma.)
Mit tehetsz, ha olyan osztályt szeretnél írni, amelyből csak EGY példány hozható létre, hogy mindenkinek, aki ebből az osztályból szeretne egy példányt, ezt az egy példányt kelljen használnia?

Statikus változók ellátása kezdőértékkel

A statikus változók akkor kapnak kezdőértéket, amikor *az osztály betöltődik*. Egy osztály betöltésére akkor kerül sor, amikor a JVM úgy dönt, hogy ideje betölteni. A JVM általában akkor tölt be egy osztályt, amikor valaki először próbál új példányt létrehozni az osztályból, vagy először veszi használatba az osztály egy statikus tagfüggvényét vagy változóját. Programozóként te is utasíthatod a JVM-t egy osztály betöltésére, de erre valószínűleg nem lesz szükséged. Szinte minden esetben jobban jársz, ha a JVM-re bízod annak eldöntését, hogy mikor *töltsön be* egy osztályt.

A statikus változók előkészítésével kapcsolatban két dolog garantált:
Egy osztály statikus változói még az előtt kezdőértéket kapnak, hogy az osztályból bármilyen *objektumot* létre lehetne hozni.
Egy osztály statikus változói még az előtt kezdőértéket kapnak, hogy az osztály bármelyik *statikus tagfüggvénye* lefutna.

Egy osztály valamennyi statikus változója az előtt kap kezdőértéket, hogy az osztályból objektumokat lehetne létrehozni.

```
class Player {
    static int playerCount = 0;
    private String name;
    public Player(String n) {
        name = n;
        playerCount++;
    }
}

public class PlayerTestDrive {
    public static void main(String[] args) {
        System.out.println(Player.playerCount);
        Player one = new Player("Tiger Woods");
        System.out.println(Player.playerCount);
    }
}
```

A playerCount akkor kap kezdőértéket, amikor az osztály betöltődik. Konkrétan megadtuk neki a 0 kezdőértéket, de ez nem kötelező, mivel az int változók alapértelmezett értéke amúgy is 0. A statikus változók ugyanúgy kapnak alapértelmezett értéket, mint a példányváltozók.

A bevezetett, de kezdőértéket nem kapott statikus és példányváltozók alapértelmezett értékei megegyeznek:
primitív egészek (long, short stb.): 0
primitív lebegőpontos számok (float, double): 0.0
logikai változók: false
objektumhivatkozások: null

A statikus változókat ugyanúgy lehet elérni, mint a statikus tagfüggvényeket: az osztály nevével.

A statikus változók tehát akkor kapnak kezdőértéket, amikor az osztály betöltődik. Ha nem adsz kifejezetten kezdőértéket egy statikus változónak (vagyis a bevezetésekor nem rendelsz hozzá értéket), akkor a változó egy alapértelmezett értéket kap. Az int típusú változók alapértelmezett értéke például 0, ami azt jelenti, hogy nem kell kifejezetten kimondanod, hogy „playerCount = 0". Ha bevezetsz egy statikus változót, de nem adsz neki kezdőértéket, akkor a változó a típusának megfelelő alapértelmezett értéket veszi fel, pontosan ugyanúgy, mint ahogy a példányváltozók is alapértelmezett értéket kapnak, amikor bevezetik őket.

```
File Edit Window Help What?
% java PlayerTestDrive
0
1
```

0 ← mielőtt bármilyen példány létrejönne

1 ← miután egy objektum létrejött

A statikus végleges változók állandók

Ha egy változó **final** (végleges) megjelölést kapott, akkor a létrehozása/előkészítése után soha nem változhat meg az értéke. Más szavakkal, egy statikus végleges változó értéke mindig ugyanaz marad, amíg az osztály be van töltve. Ha kikeresed a Math.PI-t az API-ból, ezt találod:

```
public static final double PI = 3.141592653589793;
```

A változó azért **public** (nyilvános) megjelölésű, hogy bármely kódrész hozzáférhessen; azért **static**, hogy ne kelljen példányt létrehozni a Math osztályból (amiből – ha még emlékszel – nem is szabad példányt létrehoznod); végül pedig azért **final**, mert a PI értéke nem változik (a Java szemszögéből).
Egy változót semmilyen más módon nem lehet állandóként meghatározni, az elnevezések egyezményes szabályai azonban segítenek, hogy felismerd, ha egy állandóval találkozol:
Az állandó változók nevét egyezményesen csupa nagybetűvel írjuk.

A statikus előkészítő (static initializer) olyan kódblokk, amely egy osztály betöltésekor fut le, mielőtt bármilyen kód használhatná az osztályt, ezért kiváló hely arra, hogy kezdőértéket adj egy statikus végleges változónak:

```
class Foo {
    final static int x;
    static {
      x = 42;
    }
}
```

Final statikus változó előkészítése

1 A változó bevezetésekor:

```
public class Foo {
   public static final int FOO_X = 25;
}
```

Figyeld meg a név egyezményes alakját: a statikus végleges változók állandók, ezért a nevüket csupa nagybetűvel kell írni, és a szavakat aláhúzásjellel elválasztani.

VAGY

2 Egy statikus előkészítőben:

```
public class Bar {
   public static final double BAR_SIGN;

   static {
       BAR_SIGN = (double) Math.random();
   }
}
```

Ez a kód rögtön lefut, amint az osztály betöltődött, mielőtt bármilyen statikus tagfüggvény meghívására sor kerülne, és mielőtt bármelyik statikus változót használatba lehetne venni.

Ha egy végleges változónak a fenti két hely egyikén sem adsz értéket…

```
public class Bar {
   public static final double BAR_SIGN;
}
```

Hiányzik az előkészítés!

…a fordító hibát jelez:

```
File Edit Window Help Jack-in
% javac Bar.java
Bar.java:1: variable BAR_SIGN
might not have been initialized
1 error
```

Nem csak a statikus változók lehetnek véglegesek...

A **final** kulcsszót nem statikus változók módosítására is használhatod, beleértve a példányváltozókat, a helyi változókat, sőt még a tagfüggvény-paramétereket is. A kulcsszó mindegyik esetben ugyanazt jelenti: az érték nem változtatható meg. A `final` kulcsszót ugyanakkor arra is használhatod, hogy megakadályozd, hogy valaki felülírjon egy tagfüggvényt, vagy alosztályt származtasson az osztályból.

nem statikus final változók

```
class Foof {
    final int size = 3;
    final int whuffie;

    Foof() {
        whuffie = 42;
    }

    void doStuff(final int x) {
        // az x értékét nem módosíthatod
    }

    void doMore() {
        final int z = 7;
        // a z értékét nem módosíthatod
    }
}
```

a méretet most nem módosíthatod

a whuffie értékét most nem módosíthatod

final tagfüggvények

```
class Poof {
    final void calcWhuffie() {
        // fontos dolgok, amelyeket
        // soha nem szabad felülírni
    }
}
```

final osztályok

```
final class MyMostPerfectClass {
    // nem bővíthető
}
```

Ha egy változó final, az azt jelenti, hogy nem változtathatod meg az értékét.

Ha egy tagfüggvény final, az azt jelenti, hogy a tagfüggvényt nem írhatod felül.

Ha egy osztály final, az azt jelenti, hogy az osztályt nem bővítheted (vagyis nem származtathatsz belőle alosztályt).



Nincsenek

hülye kérdések

K: Egy statikus tagfüggvény nem érhet el egy nem statikus változót. És egy nem statikus tagfüggvény hozzáférhet egy statikus változóhoz?

V: Természetesen. Egy osztály nem statikus tagfüggvénye mindig meghívhat az osztályból egy statikus tagfüggvényt, illetve elérheti az osztály statikus változóit.

K: Mikor lehet szükségem arra, hogy egy osztályt véglegessé tegyek? Ez nem megy szembe az objektumközpontúság céljával?

V: Igen is, meg nem is. Egy osztályt jellemzően a biztonság érdekében szoktak véglegessé tenni. A String osztályból például nem származtathatsz alosztályokat. Képzeld el, micsoda kavarodást okozna, ha valaki egyszer csak bővítené a String osztályt, és a többalakúságot kihasználva a saját String-alosztálybeli objektumait helyettesítené be oda, ahol a program String objektumokat vár. Ha egy osztály tagfüggvényeinek egy bizonyos megvalósítására kell támaszkodnod, mindig tedd az osztályt véglegessé.

K: Nem felesleges a tagfüggvényeket is final megjelöléssel ellátni, ha már maga az osztály is final?

V: Ha egy osztály final, a tagfüggvényeit nem kell final megjelöléssel ellátnod. Gondold végig: ha az osztály végleges, akkor senki nem származtathat belőle alosztályokat, így a tagfüggvényeit sem lehet felülbírálni.
Másrészt azonban, ha *mégis* meg szeretnéd engedni, hogy mások bővítsék az osztályodat, és azt szeretnéd, hogy felülírhassanak bizonyos tagfüggvényeket (de nem az összeset), akkor ne adj final megjelölést az osztálynak, hanem csak a kívánt tagfüggvényeket tedd véglegessé.
Ha egy tagfüggvény final, az azt jelenti, hogy az alosztályok az említett tagfüggvényt nem írhatják felül.

Újratöltve

- A ***statikus tagfüggvényeket*** az osztály, és nem egy objektumra hivatkozó változó nevével kell meghívni: `Math.random()` vs. `myFoo.go()`.
- A statikus tagfüggvények meghívásához nem kell, hogy a kupacban legyen egy példány a tagfüggvény osztályából.
- A statikus tagfüggvények olyan segédfüggvénynek alkalmasak, amely nem függ (és soha nem is fog függni) egy konkrét példányváltozó értékétől.
- A statikus tagfüggvények nem konkrét példányokhoz, hanem csak az osztályhoz kapcsolódnak, így nem érhetik el az osztályuk egyetlen példányváltozójának az értékét sem, ugyanis nem tudnák, *melyik* példány értékét kell használniuk.
- A statikus tagfüggvények nem férhetnek hozzá a nem statikus tagfüggvényekhez, mivel a nem statikus tagfüggvények általában példányváltozókban tárolt állapotjellemzőkre támaszkodnak.
- Ha egy osztályban csak statikus tagfüggvények találhatók, és az osztályt nem szeretnéd példányosítani, tedd a konstruktort priváttá.
- A ***statikus változók*** olyan változók, amelyeken az adott osztály összes tagja osztozik. A statikus változókból egy osztályban csak egy példány létezik, nem pedig példányváltozónként egy.
- A statikus tagfüggvények hozzáférhetnek a statikus változókhoz.
- A Javában úgy hozhatsz létre állandót, hogy egy változót egyszerre jelölsz meg statikusként (static) és véglegesként (final).
- A statikus végleges változóknak kötelező vagy a bevezetésükkor, vagy egy statikus előkészítőben értéket adni:

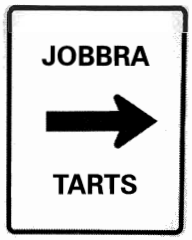
```
static {
    DOG_CODE = 420;
}
```

- Az állandók (statikus végleges változók) nevét egyezményesen csupa nagybetűvel írjuk.
- A végleges változók értékét nem lehet megváltoztatni, miután kezdőértéket kaptak.
- A végleges *példány*változóknak vagy a bevezetésükkor, vagy a konstruktorban kell értéket adni.
- A végleges tagfüggvények nem írhatók felül.
- A végleges osztályok nem bővíthetők (vagyis nem lehet alosztályokat származtatni belőlük).

Hegyezd ki a ceruzád!

Melyik érvényes?

Annak alapján, amit az imént a static és a final elemekről tanultál, az alábbi kódok közül melyek fordíthatók le?



1
```
public class Foo {
   static int x;

   public void go() {
      System.out.println(x);
   }
}
```

2
```
public class Foo2 {
  int x;

  public static void go() {
      System.out.println(x);
  }
}
```

3
```
public class Foo3 {
  final int x;

  public void go() {
      System.out.println(x);
  }
}
```

4
```
public class Foo4 {
  static final int x = 12;

  public void go() {
      System.out.println(x);
  }
}
```

5
```
public class Foo5 {
  static final int x = 12;

  public void go(final int x) {
      System.out.println(x);
  }
}
```

6
```
public class Foo6 {
  int x = 12;

  public static void go(final int x) {
      System.out.println(x);
  }
}
```

A Math osztály tagfüggvényei

Most, hogy tudod, hogyan működnek a statikus tagfüggvények, szemügyre vehetünk néhányat a Math osztályból. Amit itt bemutatunk, az persze nem az összes statikus tagfüggvény, csak a legérdekesebbek. A többit – sqrt(), tan(), ceil(), floor(), asin() stb. – megtalálod az API-ban.

Math.random()

Egy double értéket ad vissza 0.0 és 1.0 között (az utóbbit nem beleértve).

```
double r1 = Math.random();
int r2 = (int) (Math.random() * 5);
```

Math.abs()

Egy double értéket ad vissza, ami az argumentum abszolút értéke. A tagfüggvény túlterhelt, így ha egy int értéket adsz át neki, akkor int-et is ad vissza, míg ha double-t, akkor double-t.

```
int x = Math.abs(-240); // 240-et ad vissza
double d = Math.abs(240.45); // 240.45-öt ad vissza
```

Math.round()

Egy int vagy egy long értéket ad vissza (attól függően, hogy az argumentum float vagy double típusú), a legközelebbi egész értékre kerekítve.

```
int x = Math.round(-24.8f); // -25-öt ad vissza
int y = Math.round(24.45f); // 24-et ad vissza
```

Ne feledd, a literális lebegőpontos értékeket a Java double típusúnak tekinti, hacsak nem adod hozzájuk az „f" jelölést.

Math.min()

A két argumentuma közül a kisebbet adja vissza. A tagfüggvény túlterhelt, így int, long, float és double típusú értékeket is elfogad.

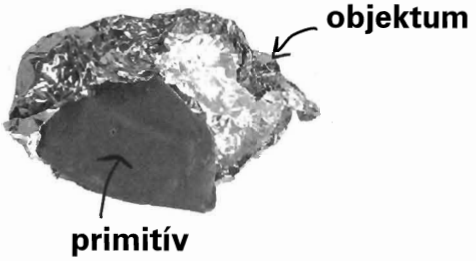
```
int x = Math.min(24,240); // 24-et ad vissza
double y = Math.min(90876.5, 90876.49); // 90876.49-et ad vissza
```

Math.max()

A két argumentuma közül a nagyobbat adja vissza. A tagfüggvény túlterhelt, így int, long, float és double típusú értékeket is elfogad.

```
int x = Math.max(24,240); // 240-et ad vissza
double y = Math.max(90876.5, 90876.49); // 90876.5-öt ad vissza
```

Primitívek beburkolása



Néha előfordul, hogy egy primitívet objektumként szeretnél kezelni. A Java 5.0 előtti változataiban például egy primitívet nem lehetett közvetlenül egy olyan gyűjteménybe helyezni, mint az ArrayList vagy a HashMap:

```
int x = 32;
ArrayList list = new ArrayList();
list.add(x);
```

Ez nem működik, hacsak nem a Java 5.0-s vagy későbbi változatát használod! Az ArrayList-ben nincs int-et elfogadó add(int) tagfüggvény! (Az ArrayList-nek csak olyan add() tagfüggvényei vannak, amelyek objektumhivatkozásokat várnak, nem primitíveket.)

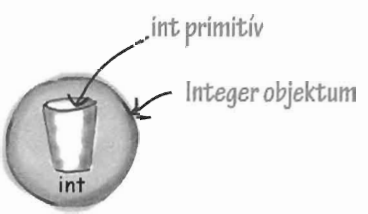
Minden primitív típushoz létezik burkolóosztály, és mivel a burkolóosztályok a java.lang csomagban találhatók, nem kell betöltened őket. A burkolóosztályokat arról ismerheted fel, hogy a nevük az általuk beburkolt primitív típust követi, de az első betű – az osztályok egyezményes elnevezési szabályainak megfelelően – nagybetű.

Ja, és bizonyos – e bolygón senki által nem ismert – okokból kifolyólag az API tervezői úgy döntöttek, hogy az osztálytípusok neveit nem illesztik pontosan a primitív típusokéhoz. Mindjárt látni fogod, mire gondolunk:

Ha egy primitívet objektumként kell kezelned, burkold be. Ha a Java 5.0-nál korábbi változatot használsz, ezt kell tenned, ha egy primitív értéket egy olyan gyűjteményben szeretnél tárolni, mint az ArrayList vagy a HashMap.

Boolean
Character
Byte
Short
Integer
Long
Float
Double

Vigyázat! A nevek nem követik pontosan a primitív típusokét: az osztálynevekben nincs rövidítés.



Integer objektum

érték beburkolása

A primitívet átadjuk a burkoló konstruktorának. Ennyi az egész.

```
int i = 288;
Integer iWrap = new Integer(i);
```

Minden burkoló így működik. A Boolean osztálynak booleanValue() tagfüggvénye van, a Character-nek charValue(), és így tovább.

érték kicsomagolása

```
int unWrapped = iWrap.intValue();
```

Megjegyzés: a lap tetején látható kép egy alufóliába burkolt csokidarabot ábrázol. Érted, „burkolt". Egyesek a képen inkább egy fóliában sütött krumplit látnak, de a párhuzam úgy is működik.



A Java 5.0 előtt NEKED kellett elvégezned a munkát...

A lánynak igaza van. A Java 5.0 előtt a primitívek a Java valamennyi változatában primitívek voltak, az objektumhivatkozások pedig objektumhivatkozások, és SOHA nem lehetett felcserélni őket. A burkolást és a kicsomagolást mindig neked, a programozónak kellett elvégezned. Egy objektumhivatkozást váró tagfüggvénynek semmilyen módon nem lehetett átadni egy primitívet, és egy objektumhivatkozást visszaadó tagfüggvény eredményét sem volt mód közvetlenül egy primitív változóhoz rendelni – még akkor sem, ha a visszaadott hivatkozás egy Integer-re mutatott, és a primitív változó int típusú volt. Egy Integer és egy int között egyszerűen nem volt semmilyen kapcsolat, attól eltekintve, hogy az Integer-nek volt egy int típusú példányváltozója (az Integer által beburkolt primitív tárolására). A munkát teljes egészében neked kellett elvégezned.

Primitív egészek tömblistája

automatikus dobozolás *nélkül* (a Java 5.0 előtt)

```
public void doNumsOldWay() {

    ArrayList listOfNumbers = new ArrayList();

    listOfNumbers.add(new Integer(3));

    Integer one = (Integer) listOfNumbers.get(0);

    int intOne = one.intValue();
}
```

Létrehozol egy tömblistát. (Ne feledd, az 5.0-s változat előtt nem határozhattad meg a TÍPUST, ezért minden ArrayList Object-ek tömbje volt.)

A „3" primitívet nem adhatod a listához; először be kell burkolnod egy Integer objektumba.

Object típusúként távozik, de az Object-et Integer-ré alakíthatod.

Végül kinyerheted a primitívet az Integer objektumból.

Automatikus dobozolás: a határ elmosódik primitív és objektum között

A Java 5.0-ban megjelent automatikus dobozolás (autoboxing) *önműködően* gondoskodik a primitívek burkolóobjektumokká alakításáról.
Lássuk, mi történik, ha egy int-eket tároló ArrayList-et szeretnél létrehozni:

Primitív egészek tömblistája

***automatikus dobozolással* (a Java 5.0-s és későbbi változataiban)**

```
public void doNumsNewWay() {

    ArrayList<Integer> listOfNumbers = new ArrayList<Integer>();

    listOfNumbers.add(3);

    int num = listOfNumbers.get(0);
}
```

Létrehozol egy Integer típusú tömblistát.

Csak add hozzá!

Bár az ArrayList-ben NINCS add(int) tagfüggvény, a fordító elvégzi helyetted a beburkolást (dobozolást). Más szavakkal, VALÓJÁBAN egy Integer objektum tárolódik az ArrayList-ben, de „úgy tehetsz", mintha a tömblista int-eket fogadna. (Egy ArrayList<Integer>-hez int-eket és Integer objektumokat egyaránt hozzáadhatsz.)

A fordító ezenkívül önműködően kicsomagolja (kiveszi a dobozból) az Integer objektumot, így az int értéket közvetlenül hozzárendelheted egy primitívhez, anélkül, hogy meg kellene hívnod az intValue() tagfüggvényt az Integer objektumra.

K: Miért nem vezetünk be egy ArrayList<int>-et, ha int-eket szeretnénk tárolni?

V: Mert *nem lehet.* Ne felejtsd el, hogy az általános típusokra vonatkozó szabály az, hogy csak osztály- vagy felülettípust határozhatsz meg, *primitív típust nem.* Az ArrayList<int> tehát nem fordítható le. Amint azonban a fenti kódból láthatod, ez nem igazán számít, mivel a fordító megengedi, hogy int-eket tegyél az ArrayList<Integer>-be. Valójában nincs is igazán semmi, ami megakadályozhatna abban, hogy primitíveket tegyél egy olyan tömblistába, amelynek a típusa megegyezik a primitív burkolójának típusával, ha a Java 5.0-nak megfelelő fordítót használsz, mivel az automatikusan dobozol. Egy ArrayList<Boolean>-ba tehát tehetsz boolean típusú primitíveket, egy ArrayList<Character>-be pedig char-okat.

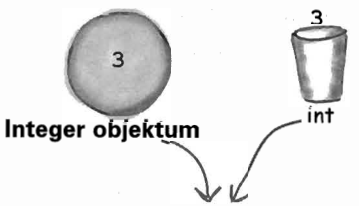
Az automatikus dobozolás szinte mindenhol működik

Az automatikus dobozolás nem csak arra jó, ami nyilvánvaló, vagyis be- és kicsomagolásra, hogy primitíveket használhass egy gyűjteményben – azt is lehetővé teszi, hogy egy primitívet lényegében bárhol felcserélhess a burkolótípusával, és viszont, ahol a program az egyiket vagy a másikat várja. Gondold végig, milyen hasznos ez!

Az automatikus dobozolás izgalmas

Tagfüggvény-argumentumok

Ha egy tagfüggvény egy burkolótípust vár, akkor átadhatsz neki egy burkolóra mutató hivatkozást és egy megfelelő típusú primitívet is. Ennek természetesen a fordítottja is igaz: ha egy tagfüggvény primitívet vár, akkor egy megfelelő primitívet, illetve az adott primitív típus burkolójára mutató hivatkozást is átadhatsz neki.

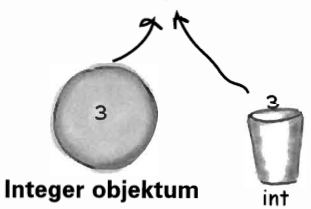



```
void takeNumber(Integer i) { }
```

Visszatérési értékek

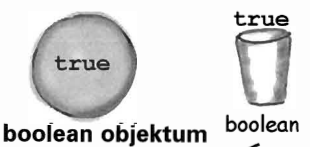
Ha egy tagfüggvény primitív visszatérési típust jelent be, egy megfelelő primitívet, illetve az adott primitív típus burkolójára mutató hivatkozást adhatsz vissza. Ha pedig a tagfüggvény burkoló visszatérési típust határoz meg, a visszaadott érték egy a burkolótípusra mutató hivatkozás vagy a megfelelő típusú primitív lehet.

```
int giveNumber() {
  return x;
}
```



Logikai kifejezések

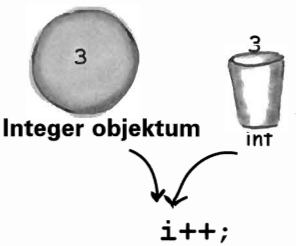
Bárhol, ahol a program boolean értéket vár, használhatsz logikai értéket eredményező kifejezést (például 4 > 2), boolean primitívet, és Boolean burkolóra mutató hivatkozást is.




```
if (bool) {
    System.out.println("true");
}
```

Számokon végzett műveletek

Valószínűleg ez a legfurcsább – bizony, most már burkolótípust is használhatsz tényezőként olyan műveletekben, ahol a program primitív típust vár. Ez azt jelenti például, hogy a növelő műveletet egy Integer objektumra mutató hivatkozásra is alkalmazhatod! De ne aggódj: ez csak a fordítóprogram egyik trükkje. A nyelvet nem módosították úgy, hogy a műveletek objektumokkal működjenek. A fordító egyszerűen átalakítja az objektumot a primitív típusára a művelet előtt. (Ettől még persze elég furán fest a dolog.)

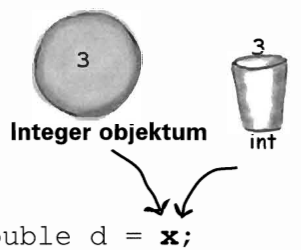



```
Integer i = new Integer(42);
i++;

Integer j = new Integer(5);
Integer k = j + 3;
```

Értékadás

Egy burkolóként vagy primitívként bevezetett változónak egy megfelelő burkolót és egy megfelelő primitívet is értékül adhatsz. Egy int típusú primitív változót például hozzárendelhetsz egy Integer típusú hivatkozó változóhoz, és fordítva, egy Integer objektumra mutató hivatkozást is értékül adhatsz egy int típusú primitívként bevezetett változónak.



Hegyezd ki a ceruzád!

Lefordítható ez a kód? Futtatható? Ha lefut, akkor mit csinál? Szánj rá időt, és gondold végig ezeket a kérdéseket, mert olyasmire világítanak rá az automatikus dobozolással kapcsolatban, amiről még nem beszéltünk.
A válaszokat a fordítóprogram segítségével derítheted ki. (Igen, arra kényszerítünk, hogy kísérletezz – természetesen a saját okulásod érdekében.)

```
public class TestBox {

   Integer i;
   int j;

   public static void main (String[] args) {
      TestBox t = new TestBox();
      t.go();
   }

   public void go() {
      j=i;
      System.out.println(j);
      System.out.println(i);
   }
}
```

Várj csak! Tovább is van! A burkolóknak is vannak statikus segédfüggvényeik!

Azon kívül, hogy úgy viselkednek, mint egy szokványos osztály, a burkolóknak van néhány igazán hasznos statikus tagfüggvényük is. Egyet már használtunk korábban: ez volt az Integer.parseInt(). Ez a feldolgozófüggvény egy String-et vár, és egy primitív értéket ad vissza.

Egy karakterlánc primitív értékké alakítása egyszerű:

A „2" könnyen 2-vé alakítható.

```
String s = "2";
int x = Integer.parseInt(s);
double d = Double.parseDouble("420.24");

boolean b = new Boolean("true").booleanValue();
```

Az 1.5-ös változatban megjelent parseBoolean() tagfüggvény nem különbözteti meg a kis- és nagybetűket a String típusú argumentumában.

De ha ezzel próbálkozol...

```
String t = "two";
int y = Integer.parseInt(t);
```

Ó, jaj. Ez rendben lefordítható, de futáskor bekrepál. Minden, ami nem értelmezhető számként, NumberFormatException kivételt okoz.

...akkor futásidejű kivételt kapsz:

```
File Edit Window Help Clue
% java Wrappers
Exception in thread "main"
java.lang.NumberFormatException: two
at java.lang.Integer.parseInt(Integer.java:409)
at java.lang.Integer.parseInt(Integer.java:458)
at Wrappers.main(Wrappers.java:9)
```

Minden karakterláncot feldolgozó tagfüggvény vagy konstruktor kiválthat NumberFormatException kivételt. Ez egy futásidejű kivétel, ezért nem kell kezelned vagy bevezetned – de esetleg szükség lehet rá. (A kivételekről a következő fejezetben lesz szó.)

És most fordítva... primitív szám átalakítása karakterlánccá

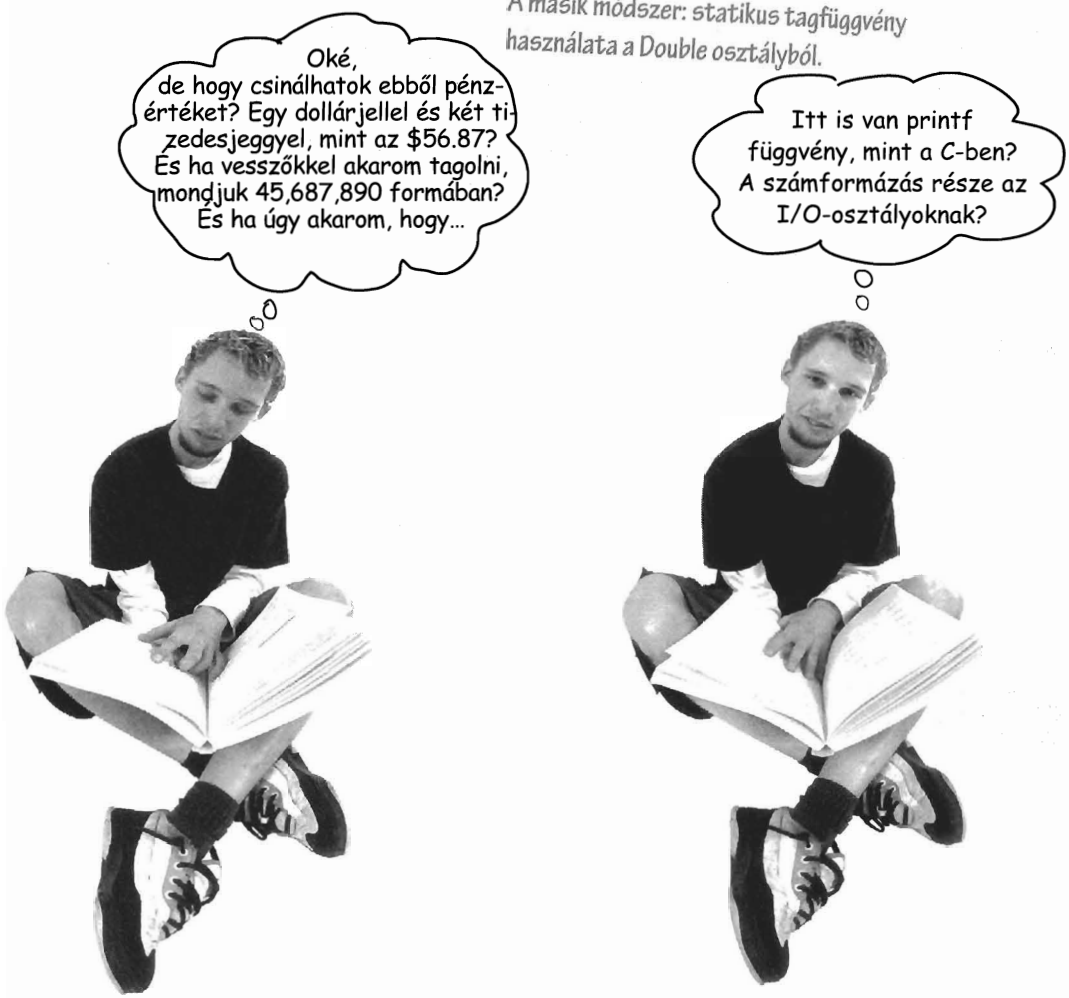
Egy számot többféleképpen is karakterlánccá lehet alakítani. A legkönnyebb módszer, ha a számot egyszerűen hozzáfűzöd egy meglevő String-hez:

```
double d = 42.5;
String doubleString = "" + d;
```

Emlékezz vissza, hogy a „+" művelet a Javában túlterhelt (ez az egyetlen túlterhelt művelet), hogy karakterláncokat is össze tudjon fűzni. Bármi, amit hozzáfűzöl egy karakterlánchoz, „elkarakterláncosodik".

```
double d = 42.5;
String doubleString = Double.toString(d);
```

A másik módszer: statikus tagfüggvény használata a Double osztályból.



Számformázás

A Javában a számok és dátumok formázása nem feltétlenül kötődik a be- és kimenethez. Gondold végig. A számok megjelenítésének egyik legszokványosabb módja egy grafikus felhasználói felület használata: karakterláncokat helyezel el egy gördülő szövegmezőben, esetleg egy táblázatban. Ha a formázás csak a kiíró utasításokba lenne beleépítve, egy számot soha nem tudnál szép karakterlánccá alakítani, amelyet megjeleníthetsz egy grafius felületen. A Java 5.0 előtt a formázás nagyját a java.text csomag olyan osztályai végezték, amelyeket ebben a könyvben egy pillantásra sem méltatunk, mivel a helyzet azóta megváltozott.

A Java 5.0-ba a Java fejlesztői további hatékony és rugalmas formázási lehetőségeket építettek be, a java.util csomag Formatter osztályán keresztül. Neked azonban nem kell tagfüggvényeket írnod és meghívnod a Formatter osztályra, mert a Java 5.0 kényelmi tagfüggvényekkel bővíti az I/O-osztályokat (beleértve a printf() függvényt is), illetve a String osztályt. Tehát csak annyit kell tenned, hogy meghívod a statikus String.format() tagfüggvényt, és átadod neki a formázandó elemet, valamint a formázási utasításokat.

A formázási utasítások megadásának módját természetesen tudnod kell, ami némi tanulást igényel, hacsak nem ismered jól a C/C++ nyelvek ***printf()*** függvényét. Szerencsére azonban akkor sincs baj, ha *nem* ismered a printf() függvényt – egyszerűen követheted az ebben a fejezetben a legegyszerűbb elemekre bemutatott recepteket. A formázást ugyanakkor *érdemes* megtanulnod, hogy a különféle lehetőségek vegyítésével *bármit* előállíthass, amit csak szeretnél.

Lássunk először egy egyszerű példát, aztán megnézzük, hogyan működik. (Megjegyzés: a formázásra még visszatérünk a be- és kimenetről szóló fejezetben.)

Számok tagolása vesszőkkel

```
public class TestFormats {

   public static void main (String[] args) {

       String s = String.format("%, d", 1000000000);
       System.out.println(s);
   }
}
```

A formázandó szám (azt szeretnénk, hogy vesszőkkel tagolt legyen).

A második argumentum (ami esetünkben egy int érték) formázásának módját leíró utasítások. Ne feledd, hogy itt csak két argumentuma van ennek a tagfüggvénynek – az első vessző a String literál BELSEJÉBEN található, tehát nem a formázó tagfüggvény argumentumait választja el.

```
1,000,000,000
```

A szám most már vesszőkkel tagova jelenik meg.

A formázás működése

Az alapokra egyszerűsítve a formázás két fő részből áll (valójában több is van, de így világosabb lesz):

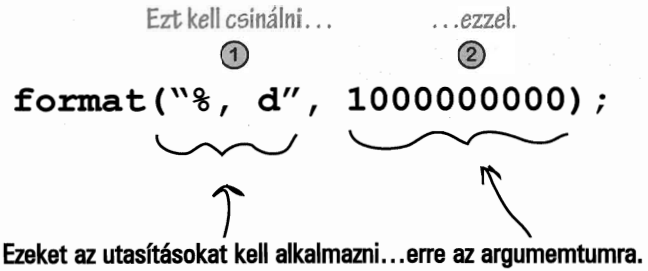
1 Formázási utasítások

Különleges formátumleírókat használhatsz, amelyek megadják, hogyan kell formázni az argumentumot.

2 A formázandó argumentum

Bár egynél több argumentum is lehet, mi csak eggyel kezdünk. Az argumentum típusa *bármi* lehet, de olyasminek kell lennie, ami formázható a formázási utasítások között megadott formátumleírókkal (format specifier). Ha a formázási utasításaid például egy lebegőpontos számot adnak meg, nem adhatsz át argumentumként egy Kutyát, sőt egy lebegőpontos számnak látszó karakterláncot sem.

Megjegyzés: ha már ismered a printf() függvényt a C vagy a C++ nyelvből, a következő néhány oldalt elég csak átfutnod. Ha nem, figyelj jól!



Mit mondanak valójában ezek az utasítások?

„Vedd a tagfüggvény második argumentumát, formázd **d**ecimális egészként, és tagold **vesszőkkel**."

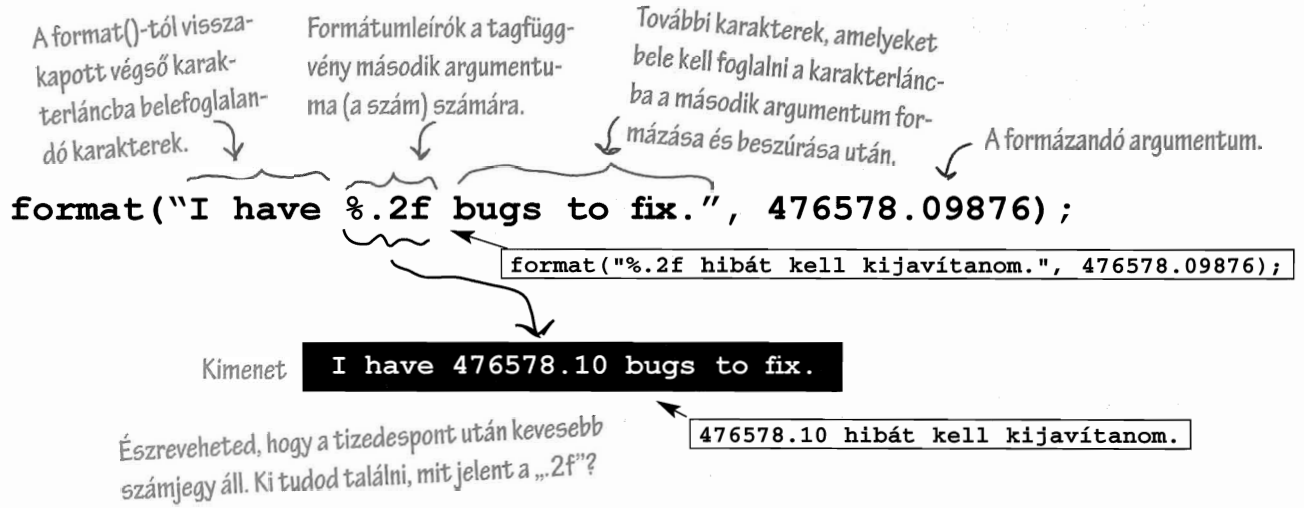
Hogyan mondják ezt?

A következő oldalon részletesebben megnézzük, mit jelent a „%, d" utasítás, de kezdetnek nem árt, ha tudod, hogy ha százalékjelet (%) látsz egy formázó karakterláncban (ami mindig az első argumentuma a format() tagfüggvényeknek), akkor gondolj rá úgy, mintha egy változót jelképezne, és a változó lenne a tagfüggvény másik argumentuma. A százalékjel utáni többi karakter az argumentum formázását vezérlő utasításokat írja le.

A százalékjel (%) jelentése: „szúrd be ide az argumentumot"

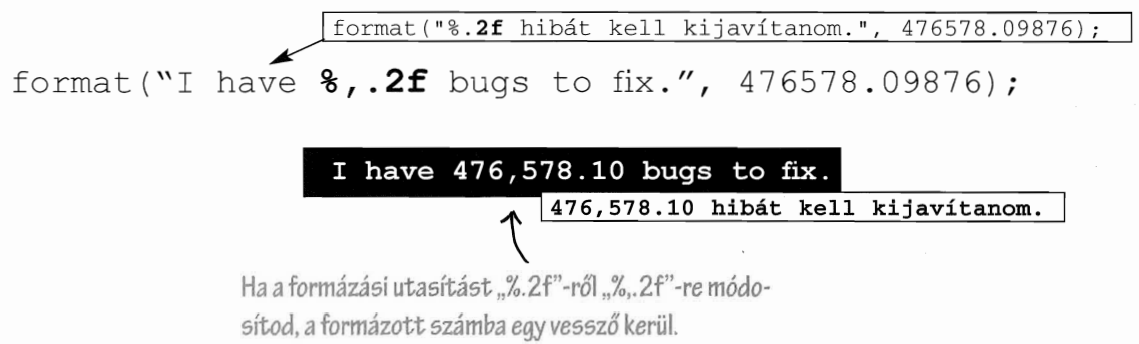
(és formázd a következő utasításokkal)

A format() tagfüggvény első argumentumát formázó karakterláncnak hívják, és olyan karakterek is lehetnek benne, amelyeket úgy szeretnél kiíratni, ahogy vannak, további formázás nélkül. Amikor azonban a százalékjelet látod, gondolj rá úgy, mint egy változóra, ami a tagfüggvény másik argumentumát jelképezi.



A „%" jel arra utasítja a formázót, hogy szúrja be a másik tagfüggvény-argumentumot (a format() második argumentumát, vagyis a számot) a % helyére, ÉS formázza a százalékjel után álló „.2f" karaktereknek megfelelően. Ezt követően a formázó karakterlánc maradéka („hibát kell kijavítanom") hozzáadódik a végső kimenethez.

Vessző hozzáadása





A formázó karakterláncnak saját nyelvtana van

Nyilvánvaló, hogy nem tehetsz *akármit* a százalékjel után. A százalékjel utáni rész különleges nyelvtani szabályokat követ, és azt írja le, hogyan kell formázni a végeredményként kapott (formázott) karakterlánc adott pontjára beszúrandó argumentumot.

Néhány példát már láttál:
A **%, d** jelentése: „szúrj be vesszőket, és formázd a számot decimális egészként".
A **%.2f** jelentése: „formázd a számot lebegőpontos számként, két tizedesjegy pontossággal".
A **%,.2f** jelentése: „szúrj be vesszőket, és formázd a számot lebegőpontos számként, két tizedesjegy pontossággal".

A valódi kérdés ez: „Honnan tudhatom, hogy mit kell tenni a százalékjel után, hogy a függvény azt csinálja, amit szeretnék?". Ehhez ismerned kell a szimbólumok jelentését – például „d" mint decimal (decimális), és „f" mint floating point (lebegőpontos) –, és a sorrendet is, amelyben az utasításokat el kell helyezned a százalékjel után. Ha a vesszőt például „%,d" helyett „%d" formában teszed a „d" után, az utasítás nem fog működni!

Vagy mégis? Mit gondolsz, mit eredményez az alábbi utasítás?

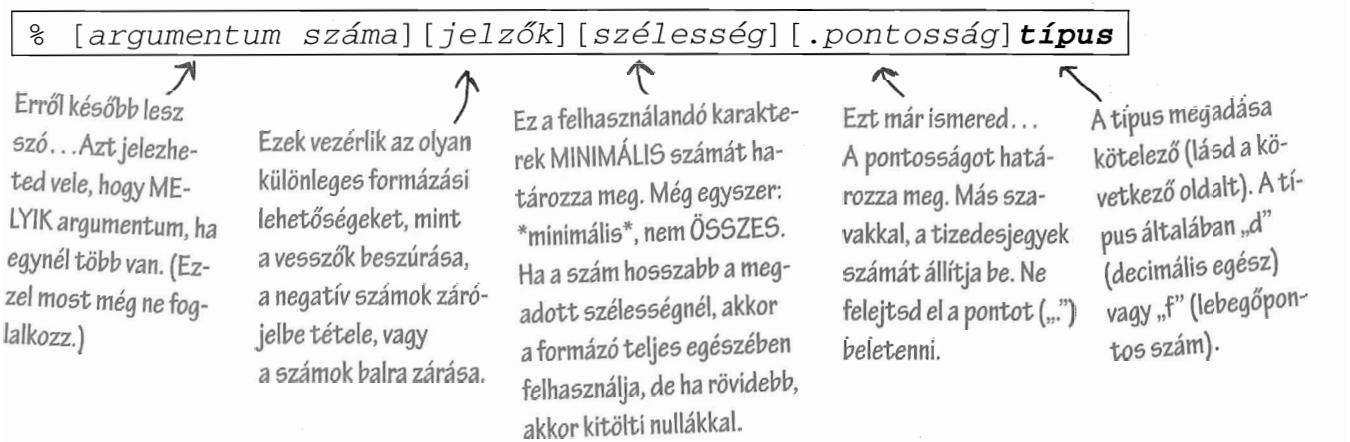
```
String.format("%.2f, hibát kell kijavítanom.", 476578.09876);
```

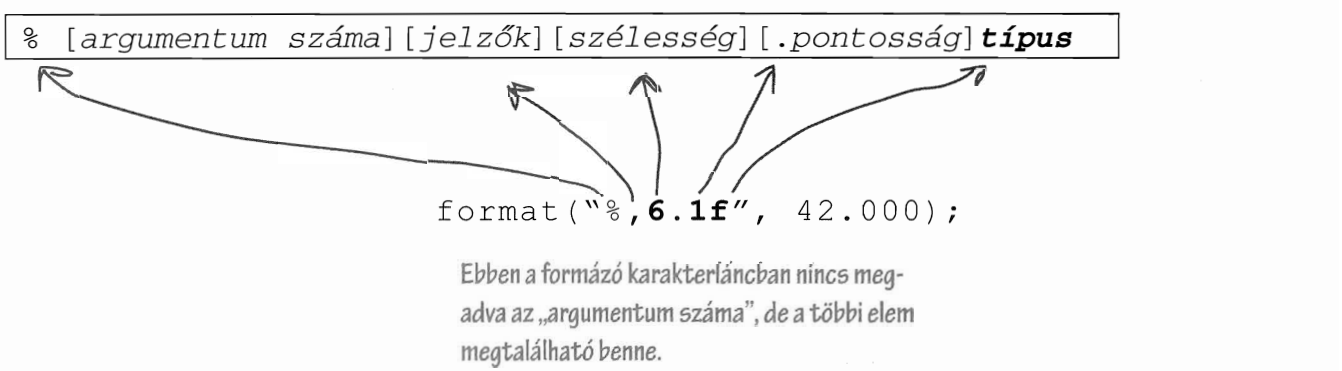
(A választ a következő oldalon találod.)

A formátumleíró

Minden, ami a százalékjel után áll, a típusjelzőt (amilyen a „d" vagy az „f") is beleértve, a formázási utasítások része. A típusjelző után a formázó feltételezi, hogy a további karakterek a kimenetre kerülő karakterlánchoz tartoznak, hacsak (vagy amíg) nem találkozik egy újabb százalékjellel. Hm. Ez egyáltalán lehetséges? Egynél több változó is lehet, ami formázandó argumentumot határoz meg? Tedd félre egy pillanatra ezt a kérdést – hamarosan visszatérünk rá. Egyelőre nézzük meg a formátumleírók nyelvtanát – vagyis azokat a dolgokat, amelyek a százalékjel után állnak, és amelyek leírják, hogyan kell formázni az argumentumot.

Egy formátumleíró akár öt részből is állhat (a százalékjelet nem számítva). Minden, amit itt szögletes zárójelek között látsz, elhagyható, vagyis csak a % és a típus megadása kötelező. A sorrend azonban kötött, tehát BÁRMELY részeket adod is meg, ebben a sorrendben kell követniük egymást.





Az egyetlen kötelező leíró a TÍPUST adja meg

Bár a típust meghatározó leíró az egyetlen, ami kötelező, ne felejtsd el, hogy ha *bármi mást* megadsz, a típusnak mindig az utolsónak kell lennie! Több mint egy tucat különböző típusmódosító létezik (és ebben a dátumok és időpontok még nincsenek is benne – ezeknek saját módosítóik vannak), de a legtöbbször valószínűleg a %d (decimális) vagy az %f (lebegőpontos) módosítót fogod használni. Ezenkívül az %f-et általában egy pontosságjelzővel kombináljuk, hogy beállítsuk a kimenetben szerepeltetni kívánt tizedeshelyek számát.

A TÍPUS megadása kötelező, minden más elhagyható.

%d ***decimal*** ***(decimális)***

```
format("%d", 42);
```

```
42
```

Egy olyan szám, mint a 42.25, nem működne. Olyan lenne, mintha közvetlenül próbálnál egy double értéket hozzárendelni egy int típusú változóhoz.

Az argumentumnak int-megfelelőnek kell lennie, vagyis csak byte, short, int vagy char (illetve ezek burkolótípusa) lehet.

%f ***floating point*** ***(lebegőpontos)***

```
format("%.3f", 42.000000);
```

```
42.000
```

Itt az „f"-et a „.3" pontosságjelzővel kombináltuk, így három nullát kaptunk.

Az argumentumnak lebegőpontos típusúnak kell lennie, vagyis csak float vagy double (primitív vagy burkoló) lehet, illetve valami, ami BigDecimal-nak hívnak (és amivel ebben a könyvben nem foglalkozunk).

%x ***hexadecimal*** ***(hexadecimális)***

```
format("%x", 42);
```

```
2a
```

Az argumentum csak byte, short, int, long (beleértve mind a primitív, mind a burkolótípusokat), illetve BigInteger típusú lehet.

%c ***character*** ***(karakter)***

```
format("%c", 42);
```

```
*
```

A 42 szám a * karaktert jelképezi.

Az argumentum csak byte, short, char vagy int lehet (beleértve mind a primitív, mind a burkolótípusokat).

A formázási utasítások között kötelező szerepelnie egy típusmeghatározásnak, és ha a típuson kívül bármi mást megadsz, mindig a típusnak kell az utolsónak lennie. A számokat az esetek többségében a „d" (decimális) vagy az „f" (lebegőpontos) leíróval formázzuk.

Mi történik, ha egynél több argumentum van?

Tegyük fel, hogy egy ilyen karakterláncot szeretnél:
„A helyezés ***20,456,654*** a ***100,567,890.24*** helyből."

A számok azonban változókból származnak. Mit kell ilyenkor tenned? Egyszerűen megadsz *két* argumentumot a formázó karakterlánc (az első argumentum) után, így a format() hívásnak kettő helyett három argumentuma lesz, az első argumentumon (a formázó karakterláncon) belül pedig két különböző formátumleíró (két olyan elem, ami százalékjellel kezdődik). Az első formátumleíró a tagfüggvény második argumentumát, a második formátumleíró pedig a tagfüggvény harmadik argumentumát szúrja be. Más szavakkal, a változók beszúrása a formázó karakterláncban azt a sorrendet követi, amelyben a további argumentumok átadódnak a format() tagfüggvénynek.

```
int one = 20456654;
double two = 100567890.248907;
String s = String.format("The rank is %,d out of %,.2f", one, two);
```

```
The rank is 20,456,654 out of 100,567,890.25
A helyezés 20,456,654 a 100,567,890.25 helyből.
```

Ha egynél több argumentum van, abban sorrendben szúródnak be, ahogy átadod őket a format() tagfüggvénynek.

Mindkét változóhoz vesszőket adtunk, és a lebegőpontos számot (a második változót) két tizedesjegyre korlátoztuk.

Ahogy látni fogod, amint a dátumformázáshoz érünk, előfordulhat, hogy ugyanarra az argumentumra különböző formátumleírókat szeretnél alkalmazni. Ezt valószínűleg nehéz elképzelni, amíg nem látod, hogyan működik a *dátumok* formázása (szemben a *számokéval*, amivel eddig foglalkoztunk). Mindjárt megtudod, hogyan fogalmazhatod meg pontosabban, hogy mely formátumleírók mely argumentumokra vonatkoznak.

K: Itt valami NAGYON furcsa. Hány argumentumot *lehet* átadni? Úgy értem, hány túlterhelt format() tagfüggvény van a String osztályban? Mi történik, ha mondjuk tíz különböző argumentumot szeretnék formázni egyetlen kimenő karakterláncban?

V: Jó kérdés. Igen, valami *valóban* furcsa (de legalábbis új és ismeretlen) dolog történik itt, és *nincs* egy csomó túlterhelt format() tagfüggvény, ami különböző számú argumentumot fogadna. Ahhoz, hogy a Java nyelv támogassa ezt az új (printf-szerű) formázási API-t, egy új szolgáltatással kellett bővíteni: a *változó argumentumlistákkal* (variable argument list, röviden *vararg*). A vararg-okról csak a függelékben esik szó, mert egy jól megtervezett rendszerben a formázáson kívül valószínűleg nem nagyon fogod másra használni őket.

Ennyit a számokról – mi a helyzet a dátumokkal?

Tegyük fel, hogy egy ilyen karakterláncot szeretnél: „Sunday, Nov 28 2004".
Semmi különös, igaz? Nos, akkor tegyük fel, hogy kezdetben csak egy Date típusú – ez egy olyan Java-osztály, amely egy időbélyeget ábrázol – változód van, és ezt az objektumot (tehát nem számot) szeretnéd átvezetni a formázón.
A számok és a dátumok formázása között a fő különbséget az jelenti, hogy a dátumformátumok egy kétkarakteres típust használnak, ami „t"-vel kezdődik (szemben az egykarakteres „f"-fel vagy „d"-vel). Az alábbi példák jól szemléltetik, hogyan is működik a dolog:

Teljes dátum és idő %tc

```
String.format("%tc", new Date());
```

```
Sun Nov 28 14:52:41 MST 2004
```

Csak az idő %tr

```
String.format("%tr", new Date());
```

```
03:01:47 PM
```

A hét napja, a hónap és a nap %tA %tB %td

Nincs egyetlen formátumleíró, ami pontosan azt eredményezné, amit szeretnénk, ezért kombinálnunk kell hármat: a hét napját (%tA), a hónapot (%tB) és a hónap napját (%td).

```
Date today = new Date();
String.format("%tA, %tB %td",today,today,today)
```

A vessző nem része a formázásnak, csupán az a karakter, amelyet ki szeretnénk íratni az első beszúrt formázott argumentum után.

```
Sunday, November 28
```

Ez azt jelenti, hogy a Date objektumot háromszor kell átadnunk, a kívánt formátum egyes részei számára. Más szavakkal, a %tA csak a hét napját adja meg, ezért utána még kérnünk kell a hónapot, aztán a hónap napját is.

Ugyanaz, mint az előző, de az argumentumok ismétlése nélkül %tA %tB %td

```
Date today = new Date();
String.format("%tA, %<tB %<td",today);
```

Ez olyasmi, mintha három különböző kiolvasó függvényt hívnál meg a Date objektumra, hogy három különböző adatot nyerj ki belőle.

A csúcsos zárójel (<) csupán egy újabb jelző a formátumleíróban, ami arra utasítja a formázót, hogy „ismét az előző argumentumot használja". Így nem kell megismételni az argumentumokat, hanem ugyanazt az argumentumot három különböző módon formázhatod.



Műveletek dátumokkal

A dátumokat általában többre használod annál, mint hogy kinyerd a *mai* dátumot. A programjaidban dátumszámításokra lehet szükség, eltelt idő kiszámítására, a feladatok fontosság szerinti ütemezésére, sőt akár teljes ütemtervek készítésére. Tehát ipari erősségű dátumkezelési lehetőségekre van szükséged.

Természetesen írhatsz saját dátumkezelő függvényeket is (ne feledkezz meg a szökőévekről – ja, és az esetleges bosszantó *szökőmásodpercekről*!), de ez elég bonyolult. A jó hír azonban az, hogy a Java API tele van olyan osztályokkal, amelyek segítenek a dátumok kezelésében. Néha túl sok is van belőlük...

Előre és hátra az időben

Tegyük fel, hogy a cégednél hétfőtől péntekig dolgoznak, és azt a feladatot kaptad, hogy határozd meg az év minden hónapjának utolsó munkanapját.

Úgy tűnik, hogy a java.util.Date kissé...elavult

Korábban a java.util.Date osztályt használtuk a mai nap kiderítésére, tehát logikusnak tűnik, hogy ebben az osztályban kezdd keresni a dátumkezelési lehetőségeket. Ha azonban az API mélyére ásol, kiderül, hogy a Date tagfüggvényeinek többsége már elavult.

A Date osztály továbbra is nagyszerűen alkalmas egy „időbélyeg" kinyerésére, vagyis egy objektuméra, amely az aktuális dátumot és időt ábrázolja, így abban az esetben nyugodtan használhatod, ha a „MOST"-ra van szükséged. Szerencsére az API kínál egy másik osztályt is: más feladatokra a **java.util.Calendar**-t ajánlja.

A „most" időbélyegéhez elég a Date, de minden másra használd a Calendar osztályt.

A dátumokkal végzett műveletekhez használd a java.util.Calendar osztályt!

A Calendar (Naptár) API tervezői – szó szerint – globálisan gondolkodtak. Az alapelv az, hogy ha dátumokkal akarsz dolgozni, akkor kérsz egy naptárat (a Calendar osztály egy statikus tagfüggvényén keresztül, amelyet a következő oldalon láthatsz), a JVM pedig visszaad neked egy példányt a Calendar egy konkrét alosztályából. (A Calendar valójában elvont osztály, ezért mindig egy konkrét alosztályával dolgozol.)

Ami azonban igazán érdekes, hogy mindig a *helyi beállításoknak* (locale, „lokál") *megfelelő fajta* naptárat kapod vissza. A világ nagy része a Gergely-naptárat használja, de ha olyan helyen laksz, ahol nem ez a helyzet, beszerezhetsz Java-könyvtárakat a többi – buddhista, muzulmán, japán stb. – naptár kezelésére is.

A szabványos Java API-ba a java.util.GregorianCalendar osztályt építették be, ezért mi is ezt fogjuk használni. A legtöbb esetben azonban egyáltalán nem is kell foglalkoznod azzal, hogy a Calendar-nak melyik alosztályát használod; elég, ha a Calendar osztály tagfüggvényeire összpontosítasz.

A Calendar osztályt bővítő objektum létrehozása

Hogyan juthatsz hozzá egyáltalán egy elvont osztály „példányához"? Természetesen sehogy. Ez nem fog működni:

Ez NEM működik:

```
Calendar cal = new Calendar();
```

A fordító ezt nem engedi meg!

Helyette használd a statikus getInstance() tagfüggvényt:

```
Calendar cal = Calendar.getInstance();
```

Ennek az utasításformának más ismerősnek kell lennie – egy statikus tagfüggvényt hívunk meg.



A Calendar-ból nem kaphatsz példányt, de annak valamelyik konkrét alosztályából igen.

A Calendar-ból nyilvánvalóan nem kaphatsz példányt, mert a Calendar osztály elvont. Statikus tagfüggvényeket azonban szabadon meghívhatsz rá, mivel a *statikus* tagfüggvényeket az *osztályra*, és nem egy adott példányra alkalmazzuk. Tehát meghívod a statikus getInstance() tagfüggvényt a Calendar-ra, és visszakapsz egy…példányt az egyik konkrét alosztályból. Valamit, ami a Calendar-t bővíti (ami azt jelenti, hogy a többalakúság révén hozzárendelhető a Calendar-hoz), és ami – a „szerződése" szerint – képes válaszolni a Calendar osztály tagfüggvényeinek.

A világ nagy részén és alapértelmezés szerint a Java legtöbb változatában egy **java.util.GregorianCalendar**-példányt fogsz kapni.

A Calendar objektumok használata

Néhány fontos dologgal tisztában kell lenned ahhoz, hogy Calendar objektumokkal dolgozhass:

- **A mezők *állapotot* tárolnak** – A Calendar objektumoknak számos mezőjük van, amelyek a dátum és idő (az objektum állapota) különféle jellemzőit ábrázolják. Egy Calendar objektumban kiolvashatod és beállíthatod például az *évet* (year) vagy a *hónapot* (month).
- **A dátumok és időpontok *növelhetők*** – A Calendar osztály tagfüggvényei lehetővé teszik, hogy a különböző mezőkhöz értéket adj, illetve azokból értékeket vonj ki. Például: „növeld eggyel a hónapot" vagy „vonj ki három évet".
- **A dátumok és időpontok *ezredmásodpercben* is ábrázolhatók** – A Calendar osztály lehetővé teszi, hogy a dátumokat ezredmásodpercben fejezd ki, illetve hogy egy ezredmásodpercben (egészen pontosan az 1970. január 1. óta eltelt ezredmásodpercek számával) ábrázolt dátumot visszaalakíts. Ennek köszönhetően pontosan kiszámíthatod például, hogy „mennyi idő telt el két időpont között", vagy „hozzáadhatsz 63 órát, 23 percet és 12 másodpercet az adott időponthoz".

Példa a Calendar objektumok használatára:

```
Calendar c = Calendar.getInstance();
c.set(2004,0,7,15,40);
long day1 = c.getTimeInMillis();
day1 += 1000 * 60 * 60;
c.setTimeInMillis(day1);
System.out.println("new hour " + c.get(c.HOUR_OF_DAY));
c.add(c.DATE, 35);
System.out.println("add 35 days " + c.getTime());
c.roll(c.DATE, 35);
System.out.println("roll 35 days " + c.getTime());
c.set(c.DATE, 1);
System.out.println("set to 1 " + c.getTime());
```

Az időpontot 2004. január 7. 15:40-re állítjuk. (Vedd észre, hogy a hónapok számozása nullától indul.)

Ezt átalakítjuk sok-sok ezredmásodpercre.

Hozzáadunk egy órányi ezredmásodpercet, aztán frissítjük az időpontot. (Figyeld meg a „+=" jelet, ami annyit tesz, hogy day1 = day1 + ...)

Hozzáadunk (add) 35 napot a dátumhoz, amivel februárra ugrunk.

„Előregörgetünk" (roll) 35 napot ehhez a dátumhoz képest. Ez NEM változtatja meg a hónapot!

Itt nem növelünk, csak beállítjuk (set) a dátumot.

```
File Edit Window Help Time-Flies
new hour 16
add 35 days Wed Feb 11 16:40:41 MST 2004
roll 35 days Tue Feb 17 16:40:41 MST 2004
set to 1 Sun Feb 01 16:40:41 MST 2004
```

A kimenetből jól látható, hogyan működnek az ezredmásodpercek, a hozzáadás, a görgetés és a beállítás.

A Calendar API

Az imént láttál néhányat működés közben a Calendar osztály mezői és tagfüggvényei közül. A Calendar API azonban hatalmas, ezért csak a leggyakrabban használatos mezőket és tagfüggvényeket mutatjuk be. Ha már jól ismered ezeket, az API többi részét is könnyen az uralmad alá hajthatod.

A Calendar legfontosabb tagfüggvényei

add(int *mező*, int *mennyiség*)
Időt ad a naptár mezőjéhez, vagy időt von ki belőle.

get(int *mező*)
Visszadja a megadott naptármező értékét.

getInstance()
Egy Calendar objektumot ad vissza. A lokált meghatározhatod.

getTimeInMillis()
Az adott Calendar-ban tárolt időt adja vissza ezredmásodpercben, long típusú értékként.

roll(int *mező*, boolean up)
Időt ad vagy von ki a nagyobb mezők megváltoztatása nélkül.

set(int *mező*, int *érték*)
Beállítja a megadott naptármező értékét.

set(*év*, *hónap*, *nap*, *óra*, *perc*) (mindegyik int)
A teljes idő szokványos beállítása.

setTimeInMillis(long *ezredmásodperc*)
Beállítja egy Calendar objektum idejét a long típusú, ezredmásodperc alakban megadott idő alapján.

// további tagfüggvények...

A Calendar legfontosabb mezői

DATE / DAY_OF_MONTH
A hónap napjának kiolvasására, illetve beállítására szolgál.

HOUR / HOUR_OF_DAY
A 12 vagy 24 órás érték kiolvasására, illetve beállítására szolgál.

MILLISECOND
Az ezredmásodpercek kiolvasására, illetve beállítására szolgál.

MINUTE
A perc kiolvasására és beállítására szolgál.

MONTH
A hónap kiolvasására és beállítására szolgál.

YEAR
Az év kiolvasására és beállítására szolgál.

ZONE_OFFSET
A GMT-től ezredmásodpercben számított nyers eltérés kiolvasására, illetve beállítására szolgál.

// további mezők...

További statikusok – statikus betöltés

Újdonság a Java 5.0-ban – amiről megoszlanak a vélemények. Egyesek imádják, mások gyűlölik. A statikus betöltés lehetősége csak azt a célt szolgálja, hogy kevesebbet kelljen gépelned. Ha utálsz gépelni, lehet, hogy neked is tetszeni fog. A statikus betöltés hátránya, hogy – ha nem jársz el körültekintően – a kódot lényegesen nehezebben olvashatóvá teszi.
Az alapötlet az, hogy amikor egy statikus osztályt, statikus változót vagy felsoroló értéket (enum – erről majd később) használsz, be is töltheted, és így megspórolhatsz némi gépelést.

Óvatosan használd: A statikus betöltések nehezen olvashatóvá tehetik a kódot.

Régimódi kód:

```
import java.lang.Math;

class NoStatic {

  public static void main(String [] args) {

    System.out.println("sqrt " + Math.sqrt(2.0));

    System.out.println("tan " + Math.tan(60));

  }

}
```

Ugyanaz a kód, statikus betöltésekkel:

```
import static java.lang.System.out;
import static java.lang.Math.*;

class WithStatic {

  public static void main(String [] args) {

    out.println("sqrt " + sqrt(2.0));

    out.println("tan " + tan(60));

  }

}
```

A statikus betöltések bevezetésekor használandó utasításforma.

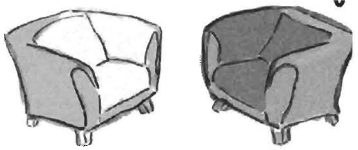
Statikus betöltések működés közben.



Buktatók és megoldások

- Ha egy statikus tagot csak néhány alkalommal fogsz használni, célszerűbb kerülni a statikus betöltést, hogy a kód áttekinthetőbb maradjon.
- Ha egy statikus tagot sokszor szándékozol használni (például sok matematikai számítást kívánsz végezni), akkor a statikus betöltés alkalmazása valószínűleg oké.
- A statikus betöltések bevezetésekor helyettesítő karaktereket (.*) is használhatsz.
- A statikus betöltések egyik nagy problémája, hogy könnyen névütközés léphet fel. Például ha két különböző osztályod is van add() nevű tagfüggvénnyel, honnan fogod tudni – és honnan tudja a fordító –, hogy melyiket kell használni?

Kandalló melletti csevely



Ma esti vendégeink, egy példányváltozó élcelődik egy statikus változón

Példányváltozó

Azt sem tudom, miért vagyunk itt. Mindenki tudja, hogy a statikus változókat csak állandónak használják. És hány is van belőlük? Ha jól tudom, az egész API-ban van talán négy. Nem mintha valaki is használná őket.

Persze, tele. Mondja csak. Oké, akkor van néhány a Swing könyvtárban, de mindenki tudja, hogy a Swing különleges eset.

Na jó, de néhány grafikus felületi dolgon kívül tud akár csak egy statikus változót is mutatni, amit valaki tényleg használ? Úgy értem, a valóságban.

Na, még egy különleges eset – amit persze amúgy sem használ senki, csak hibakereséskor.

Statikus változó

Nem ártana jobban utánanéznie a tényeknek. Mikor nézett bele utoljára az API-ba? Dugig tele van statikus elemekkel! Egész osztályokat szenteltek állandó értékek tárolásának. Van például egy SwingConstants nevű osztály, ami tele van velük.

Lehet, hogy különleges eset, de nagyon fontos! És mi a helyzet a Color osztállyal? Elég fájdalmas lenne, ha minden RGB-értékre emlékeznie kellene, hogy szabványos színeket hozzon létre! A Color osztályban azonban előre meghatározták a kék, a lila, a fehér, a vörös stb. állandóját. Nagyon kényelmes.

Kezdetnek mit szólna a System.out-hoz? Az out a System.out-ban a System osztály statikus változója. Ön személyesen nem hoz létre új példányt a System osztályból, csak kéri tőle az out változót.

Mintha a hibakeresés nem lenne fontos! És még valami, ami valószínűleg meg sem fordult a szűk agyában: lássa be, a statikus változók hatékonyabbak. Nem példányonként, hanem osztályonként van belőlük egy, ami hatalmas memóriamegtakarítást jelenthet!

Példányváltozó

Hm. Nem felejtett el valamit?

A statikus változóknál semmi sem kevésbé objektumközpontú! Ha már itt tartunk: miért nem lépünk mindjárt hátra egy hatalmas lépést, és programozunk eljárásokkal?

Maga olyan, mint egy globális változó – és minden valamirevaló programozó tudja, hogy az Nem Jó.

Igen, egy osztályban lakik, csakhogy az OO nem *osztály*központú programozást jelent. Micsoda ostobaság! Maga egy őskövület. Csak arra jó, hogy a régi motorosoknak segítsen felkapaszkodni a Java vonatára.

Persze, egyszer-egyszer biztos van értelme statikus tagot használni, de higgye el, a statikus változók (és tagfüggvények) túlzásba vitt használata az éretlen OO-programozó jele. A tervezőnek az *objektumok*, és nem az *osztályok* állapotára kell figyelnie.

A statikus tagfüggvények mind közül a legrosszabbak, mert általában azt jelzik, hogy a programozó eljárásokban gondolkodik, és nem az egyedi állapotuknak megfelelően cselekvő objektumokban.

Peeersze...győzködje csak magát, amivel akarja...

Statikus változó

Tessék?

Mit ért azon, hogy *nem* objektumközpontú?

Én NEM vagyok globális változó. Olyan nem is létezik. Én egy osztályban élek! Az elég objektumközpontú – tudja, OSZTÁLY. Nem csak lebegek valahol az űrben: természetes része vagyok egy objektum állapotának, csak annyi a különbség, hogy egy osztály minden példánya osztozik rajtam. Nagyon hatékony.

Na, most álljon le. EZ határozottan nem igaz. Bizonyos statikus változók abszolút létfontosságúak a rendszer számára. De még azok is kényelmesek, amelyek nem annyira fontosak.

Ezt miért mondja? És mi baja a statikus tagfüggvényekkel?

Természetesen tudom, hogy az objektumközpontú tervezésnek az objektumokra kell összpontosítania, de csak azért, mert van néhány szerencsétlen programozó, még nem kéne kidobni a gyereket a bájtkóddal együtt. A statikusok használatának is megvan a helye és az ideje, és amikor szükség van rájuk, semmi sem veheti fel velük a versenyt.



Légy a fordító!

Az ezen az oldalon található Java-fájl egy teljes program forrása. A te feladatod az, hogy eljátszd a fordítóprogram szerepét, és kideríted, hogy lefordítható-e a fájl. Ha nem, hogyan javítanád ki? Ha igen, milyen kimenetet eredményez?



```
class StaticSuper{

  static {
    System.out.println("super static block");
  }

  StaticSuper{
    System.out.println(
      "super constructor");
  }
}

public class StaticTests extends StaticSuper {
  static int rand;

  static {
    rand = (int) (Math.random() * 6);
    System.out.println("static block " + rand);
   }

   StaticTests() {
     System.out.println("constructor");
   }

  public static void main(String [] args) {
    System.out.println("in main");
    StaticTests st = new StaticTests();
  }
}
```

Ha a kód lefordítható, ezek közül melyik lesz a kimenet?

Lehetséges kimenet

```
File  Edit  Window  Help  Cling
%java StaticTests
static block 4
in main
super static block
super constructor
constructor
```

Lehetséges kimenet

```
File  Edit  Window  Help  Electricity
%java StaticTests
super static block
static block 3
in main
super constructor
constructor
```



Ez a fejezet feltárta előtted a Java csodálatos statikus világát. A feladatod az, hogy eldöntsd, hogy az alábbi állítások közül melyik igaz, és melyik hamis.

Igaz vagy hamis?

1. Ahhoz, hogy használhasd a Math osztályt, először létre kell hoznod belőle egy példányt.
2. Egy konstruktort megjelölhetsz a static kulcsszóval.
3. A statikus tagfüggvények nem férnek hozzá a „this" objektum példányváltozóiban tárolt állapothoz.
4. A statikus tagfüggvényeket egy hivatkozó változón keresztül ajánlott meghívni.
5. Statikus változókat használhatsz arra, hogy megszámláld egy osztály példányait.
6. A konstruktorok meghívására az előtt kerül sor, hogy a statikus változók kezdőértéket kapnának.
7. A MAX_SIZE jó név lenne egy statikus végleges változónak.
8. A statikus előkészítő blokk az osztály konstruktora előtt fut le.
9. Ha egy osztály final-ként van megjelölve, az összes tagfüggvényét is véglegessé kell tenni.
10. A final megjelölésű tagfüggvények csak akkor írhatók felül, ha az osztályukat bővítik.
11. A boolean típusú primitívekhez nem létezik burkolóosztály.
12. Burkolót akkor kell használnod, ha egy primitívet objektumként szeretnél használni.
13. A parseXxx tagfüggvények mindig karakterláncot adnak vissza.
14. A formázó osztályok (amelyek nem kapcsolódnak a be- és kimenethez) a java.format csomagban találhatók.





Holdmágnesek

Ez akár még hasznos is lehet! Az azonban, amit az utolsó néhány oldalon a dátumok kezeléséről tanultál, némi kiegészítésre szorul. Először is, telihold körülbelül 29,52 naponként van. Másodszor, 2004. január 7-én telihold volt. A feladatod az, hogy a kódrészletekből összeállítsd azt a működőképes Java-programot, amely a lap alján látható kimenetet eredményezi (illetve további telihold-dátumokat). (Lehet, hogy nem lesz szükséged valamennyi kódmágnesre, és a kapcsos zárójeleket is neked kell beírnod, ahol szükséges.) Ja, és a kimenet nálad más lesz, ha nem a Sziklás-hegységi időzónában élsz.

```
File  Edit  Window  Help  Howl
% java FullMoons
full moon on Fri Feb 06 04:09:35 MST 2004
full moon on Sat Mar 06 16:38:23 MST 2004
full moon on Mon Apr 05 06:07:11 MDT 2004
```

```
long day1 = c.getTimeInMillis();
c.set(2004,1,7,15,40);
import static java.lang.System.out;
static int DAY_IM = 60 * 60 * 24;
("full moon on %tc", c));
(c.format
Calendar c = new Calendar();
class FullMoons {
public static void main(String [] args) {
day1 += (DAY_IM * 29.52);
for (int x = 0; x < 60; x++) {
static int DAY_IM = 1000 * 60 * 60 * 24;
println
("full moon on %t", c));
import java.io.*;
import java.util.*;
static import java.lang.System.out;
c.set(2004,0,7,15,40);
out.println
c.setTimeInMillis(day1);
(String.format
Calendar c = Calendar.getInstance();
```

Gyakorlat – Megoldás

Légy a fordító!

```
StaticSuper( ) {
  System.out.println(
    "super constructor");
}
```

A StaticSuper egy konstruktor, így az aláírásában szerepelnie kell egy zárójelpárnak. Figyeld meg, hogy amint az alábbi kimenet mutatja, mindkét osztály statikus blokkja a konstruktorok előtt fut le.

Lehetséges kimenet

```
File  Edit  Window  Help  Cling
%java StaticTests
super static block
static block 3
in main
super constructor
constructor
```

Igaz vagy hamis?

1. Ahhoz, hogy használhasd a Math osztályt, először létre kell hoznod belőle egy példányt. **Hamis**
2. Egy konstruktort megjelölhetsz a static kulcsszóval. **Hamis**
3. A statikus tagfüggvények nem férnek hozzá a „this" objektum példányváltozóiban tárolt állapothoz. **Igaz**
4. A statikus tagfüggvényeket egy hivatkozó változón keresztül ajánlott meghívni. **Hamis**
5. Statikus változókat használhatsz arra, hogy megszámláld egy osztály példányait. **Igaz**
6. A konstruktorok meghívására az előtt kerül sor, hogy a statikus változók kezdőértéket kapnának. **Hamis**
7. A MAX_SIZE jó név lenne egy statikus végleges változónak. **Igaz**
8. A statikus előkészítő blokk az osztály konstruktora előtt fut le. **Igaz**
9. Ha egy osztály final-ként van megjelölve, az összes tagfüggvényét is véglegessé kell tenni. **Hamis**
10. A final megjelölésű tagfüggvények csak akkor írhatók felül, ha az osztályukat bővítik. **Hamis**
11. A boolean típusú primitívekhez nem létezik burkolóosztály. **Hamis**
12. Burkolót akkor kell használnod, ha egy primitívet objektumként szeretnél használni. **Igaz**
13. A parse*Xxx* tagfüggvények mindig karakterláncot adnak vissza. **Hamis**
14. A formázó osztályok (amelyek nem kapcsolódnak a be- és kimenethez) a java.format csomagban találhatók. **Hamis**



Gyakorlat – Megoldás

```
import java.util.*;

import static java.lang.System.out;

class FullMoons {

  static int DAY_IM = 1000 * 60 * 60 * 24;

  public static void main(String [] args) {

    Calendar c = Calendar.getInstance();

    c.set(2004,0,7,15,40);

    long day1 = c.getTimeInMillis();

    for (int x = 0; x < 60; x++) {

       day1 += (DAY_IM * 29.52)

       c.setTimeInMillis(day1);

       out.println(String.format("full moon on %tc", c));

    }

  }

}
```

Megjegyzések a Holdmágnesekkel kapcsolatban:
Felfedezheted, hogy a program által kiírt dátumok némelyike el van csúszva egy nappal. A csillagászati számítások kissé bonyolultak – ha tökéletesen hajtanánk végre őket, a dolog túl összetett lenne ehhez a gyakorlathoz.

Tipp: az időzóna-eltérésekkel kapcsolatban van egy probléma, amit megpróbálhatsz megoldani. Rá tudsz jönni, mi az?

```
File Edit Window Help Howl
% java FullMoons
full moon on Fri Feb 06 04:09:35 MST 2004
full moon on Sat Mar 06 16:38:23 MST 2004
full moon on Mon Apr 05 06:07:11 MDT 2004
```

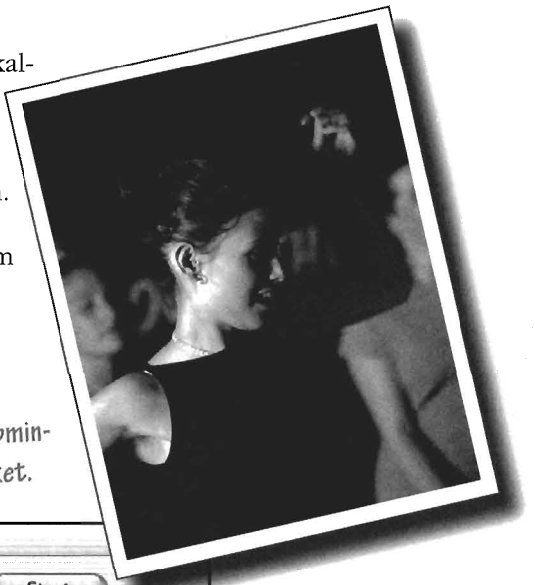
Kockázatos viselkedés



Bármi megtörténhet. A fájl nem található. A kiszolgáló lefagyott. Nem számít, milyen jó programozó vagy, nem irányíthatsz mindent. Bármi elromolhat. Akár *nagyon* elromolhat. Ha kockázatos tagfüggvényt írsz, szükséged lesz egy olyan kódrészre is, amely képes kezelni az esetleg fellépő hibákat. De honnan *tudhatod*, hogy egy tagfüggvény kockázatos-e? És hová kell tenned a ***kivételes*** helyzetet *kezelő* kódot?
A könyvben eddig nem kockáztattunk *igazán*. Voltak persze dolgok, amelyek futáskor balul ütöttek ki, de a problémák javarészt az általunk írt kód hibáiból eredtek. Programhibákból, amelyeket a fejlesztés során ki kell javítanunk. Az a problémakezelő kód azonban, amiről most beszélünk, olyan kódrészekkel foglalkozik, amelyeknek a futásidejű működését *nem tudod* szavatolni. Lehetnek például olyan kódrészek, amelyek arra számítanak, hogy egy fájl a megfelelő könyvtárban van, hogy a kiszolgáló fut, vagy hogy egy adott szál alszik. A témával pedig *most* kell foglalkoznunk, mert *ebben* a fejezetben olyasmit fogunk építeni, ami a kockázatos JavaSound API-t használja: egy MIDI-zenelejátszót.

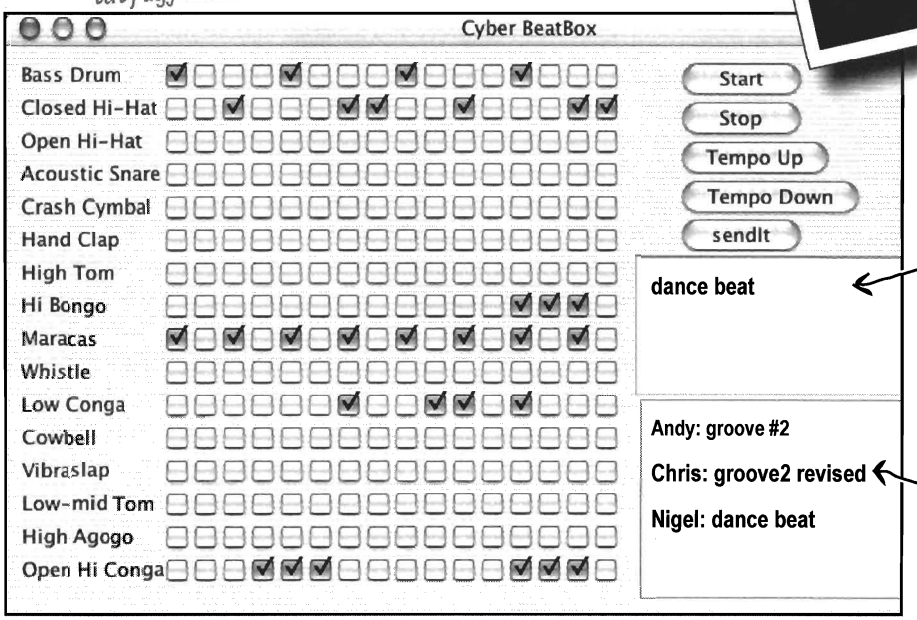
Készíts zenegépet!

A most következő három fejezet során néhány hanglejátszó alkalmazást fogunk készíteni, például egy BeatBox dobgépet. Valójában még mielőtt a könyv végére érnénk, lesz egy többszemélyes változatunk is belőle, hogy átküldhesd a dobtémáidat egy másik felhasználónak, akár egy csevegőszobában. Te írod meg az egészet, bár a grafikus felülethez használhatsz Készre sütött kódokat is, ha úgy döntenél. Jó, tudjuk, hogy nem minden IT-részleg új BeatBox-kiszolgálóra vágyik, de azért csináljuk ezt az egészet, hogy többet *tudj* meg a *Javáról*. Egy BeatBox felépítése jó szórakozás, ami *közben* tanulsz is.



A kész BeatBox valahogy így néz ki:

Beatbox-témát („loop"-ot, vagyis egy 16 ütemből álló dobmintát) úgy hozhatsz létre, hogy kipipálod a kívánt négyzeteket.



Ide jön az üzenet, amit el szeretnél küldeni a többieknek az adott dobtémával, amikor a sendit („küldd el") gombra kattintasz.

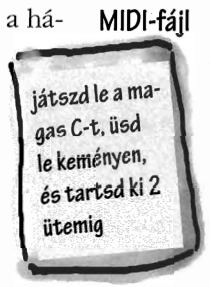
Itt jelennek meg a többiektől érkező üzenetek. Ha rákattintasz valamelyikre, betöltődik az üzenettel kapott téma, és a Start gombbal elindíthatod a lejátszását.

Pipáld ki a négyzeteket mind a 16 ütem számára. A fenti ábra szerint például az 1. ütemben a basszusdob (Bass Drum) és a maracas (Maracas) fog megszólalni, a 2. ütemben semmi nem szól, a 3. ütemben a maracas és a zárt lábcinhang (Closed Hi-Hat) – érted a lényeget. Amikor a Start gombra kattintasz, a program lejátssza a mintát, és addig ismétli (loop), amíg a Stop gombbal le nem állítod. A mintáidat bármikor „rögzítheted" azáltal, hogy elküldöd a BeatBox-kiszolgálónak (ami azt jelenti, hogy bármely másik felhasználó meghallgathatja). A beérkező mintákat a hozzájuk mellékelt üzenetre kattintva töltheted be.

Kezdjük az alapokkal!

Világos, hogy pár dolgot meg kell tanulnod, mielőtt megírhatnád a teljes programot: többek között azt, hogy miként építhetsz Swing grafikus felületet, hogyan *kapcsolódhatsz* egy másik géphez hálózaton keresztül, illetve a be- és kimenet kezeléséről is tudnod kell ezt-azt, hogy *küldhess* valamit a többi gépnek.

Na és persze meg kell ismerned a JavaSound API-t. *Ezzel* fogjuk kezdeni ebben a fejezetben. Egyelőre felejtsd el a grafikus felületet, a hálózatkezelést és az I/O-t, és összpontosíts arra, hogy MIDI-hangokat csikarj ki a számítógépedből. Ne aggódj, ha nem tudsz semmit a MIDI-ről, vagy gőzöd sincs a kottaolvasásról vagy a zeneszerzésről. Ezekben a fejezetekben mindent megtalálsz, amit tudnod kell. Szinte már a zsebedben érezheted a lemezszerződést.



A MIDI-fájl arról tartalmaz információt, hogy a dalt hogyan kell lejátszani, de tényleges hangadatok nincsenek benne, tehát olyan, mint egy kotta egy gépzongora számára.

A JavaSound API

A JavaSound olyan osztályok és felületek gyűjteménye, amelyek az 1.3-as változat óta képezik részét a Javának. Nem különleges bővítmények: a szabványos J2SE osztálykönyvtár részei. A JavaSound két részből – MIDI és Sampled – áll, de ebben a könyvben mi csak a MIDI-vel foglalkozunk. A MIDI a Musical Instrument Digital Interface (digitális hangszerfelület) rövidítése, és nem más, mint egy szabványos protokoll, amelyen keresztül a különféle elektronikus hangeszközök társaloghatnak egymással. A BeatBox alkalmazásunk esetében azonban a MIDI-re úgy is gondolhatsz, mint egyfajta kottára, amit egy high-tech „gépzongorába" táplálsz be. Más szavakkal, a MIDI-adatok ténylegesen nem tartalmaznak semmilyen *hangot*, csak *utasításokat*, amelyeket egy MIDI-képes hangszer vissza tud játszani. Vagy, ha másik hasonlatot szeretnél, a MIDI-fájlok olyanok, mint a HTML-dokumentumok, a MIDI-fájlt feldolgozó (értsd: *lejátszó*) hangszer pedig olyan, mint a webböngésző.





A MIDI-eszköz tudja, hogyan kell „olvasni" egy MIDI-fájlt és visszajátszani a hangot. Az eszköz lehet billentyűs szintetizátor, de más hangszer is. Egy MIDI-hangszer általában SOKFÉLE különböző hangot képes lejátszani (zongora, dob, hegedű stb.) – akár egyszerre. A MIDI-fájl tehát nem egyetlen hangszeres számára nyújt kottát, hanem a zenekar MINDEN tagja számára tartalmazza az adott dal lejátszásához szükséges részeket.

A MIDI-adatok meghatározzák, hogy *mit* kell csinálni (például: játszd le a középső C hangot, ilyen és ilyen hangosan, és ennyi és ennyi ideig tartsd ki stb.), de arról semmit sem mondanak, hogy ténylegesen milyen *hangot* fogsz hallani. A MIDI nem tudja, hogyan kell előállítani egy fuvola, zongora vagy Jimi Hendrix gitárjának hangzását. A hang előállításához egy (MIDI-képes) hangszerre van szükség, tehát egy olyan eszközre, amely képes elolvasni és lejátszani egy MIDI-fájlt. Az eszköz azonban inkább olyan, mint egy *egész zenekar vagy hangszerek arzenálja*, és lehet fizikai eszköz – mint a rockzenészek által használt billentyűs szintetizátorok –, de teljes egészében szoftveres úton megvalósított „hangszer" is, ami a számítógépeden lakik. A BeatBox-unk esetében csak azt a beépített, szoftveres hangszert fogjuk használni, amit a Java kínál, és amelyet *szintetizátornak* neveznek (synthesizer; egyesek *szoftveres szintetizátornak* – software synth – hívják), mert hangot *állít elő* („szintetizál") – olyan hangot, amit *meghallgathatsz*.

Először is kell egy Sequencer

Mielőtt bármilyen hangot lejátszhatnál, szükséged lesz egy Sequencer (sorozat-feldolgozó) objektumra. A sorozat-feldolgozó az az objektum, amelyik fogja az összes MIDI-adatot, és átadja azokat a megfelelő hangszereknek – tehát ez az, ami *lejátssza* a zenét. A sorozat-feldolgozók sok más dolgot is tudnak, de ebben a könyvben szigorúan csak visszajátszó eszközként használjuk a Sequencer objektumot, mintha egy CD-lejátszó lenne, csak egy kicsit több szolgáltatással. A Sequencer osztály a javax.sound.midi csomagban található (amely a Java alapkönyvtárának része az 1.3-as változat óta). Kezdd tehát azzal, hogy meggyőződsz róla, hogy létre tudsz hozni (vagy ki tudsz olvasni) egy Sequencer objektumot:

```
import javax.sound.midi.*;

public class MusicTest1 {

    public void play() {
        Sequencer sequencer = MidiSystem.getSequencer();

        System.out.println("We got a sequencer");
    } // a lejátszó függvény (play) lezárása

    public static void main(String[] args) {
        MusicTest1 mt = new MusicTest1();
        mt.play();
    } // a main lezárása
} // az osztály lezárása
```

betöltjük a javax.sound.midi csomagot

Szükségünk van egy Sequencer objektumra. Ez a fő része az általunk használt MIDI-eszköznek/hangszernek, vagyis az a dolog, ami „dallá" állítja össze a MIDI-információkat. Vadonatúj sorozat-feldolgozót azonban nem kell létrehoznunk, csak meg kell kérnünk a MidiSystem-et, hogy adjon nekünk egyet.

Valami nem stimmel!

Ez a kód nem fordítható le! A fordító azt mondja, hogy „be nem jelentett kivétel" (unreported exception) lépett fel, amit el kell fognunk, vagy be kell jelentenünk.

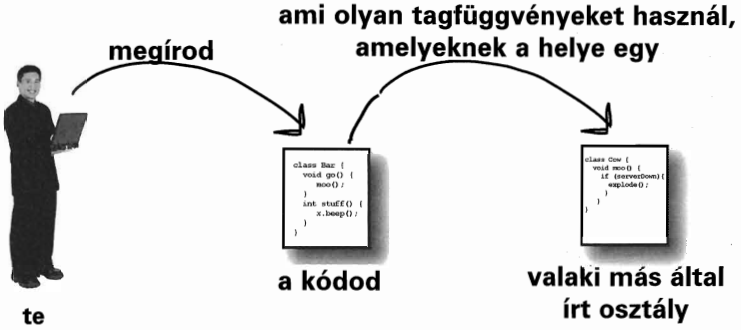
File Edit Window Help SayWhat?

```
% javac MusicTest1.java
MusicTest1.java:13: unreported exception javax.sound.midi.
MidiUnavailableException; must be caught or declared to be
thrown
    Sequencer sequencer = MidiSystem.getSequencer();
                                     ^
1 errors
```

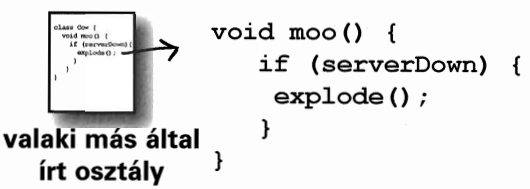
Mi van, ha *egy meghívni kívánt tagfüggvény kockázatos?*

(ami valószínűleg egy olyan osztályban található, amit nem te írtál)

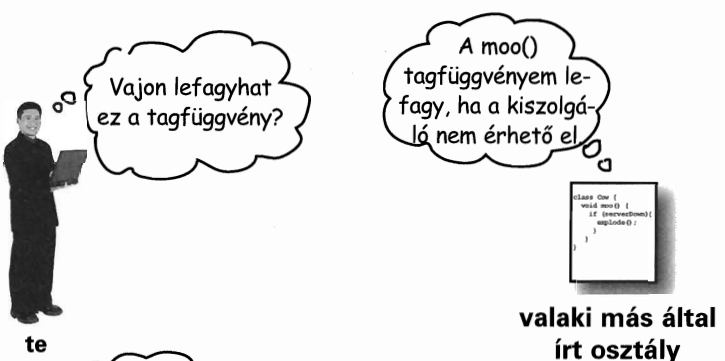
1. Tegyük fel, hogy egy olyan tagfüggvényt akarsz meghívni, amely egy olyan osztályban található, amelyet nem te írtál.



2. A tagfüggvény valamilyen kockázatos dolgot művel, ami nem biztos, hogy futásidőben működik.



3. *Tudnod* kell, hogy a meghívni kívánt tagfüggvény kockázatos.



4. Ezt követően megírhatod azt a kódot, ami képes kezelni a hibát, ha *tényleg* bekövetkezne. A biztonság kedvéért fel kell készülnöd a lehetőségre.

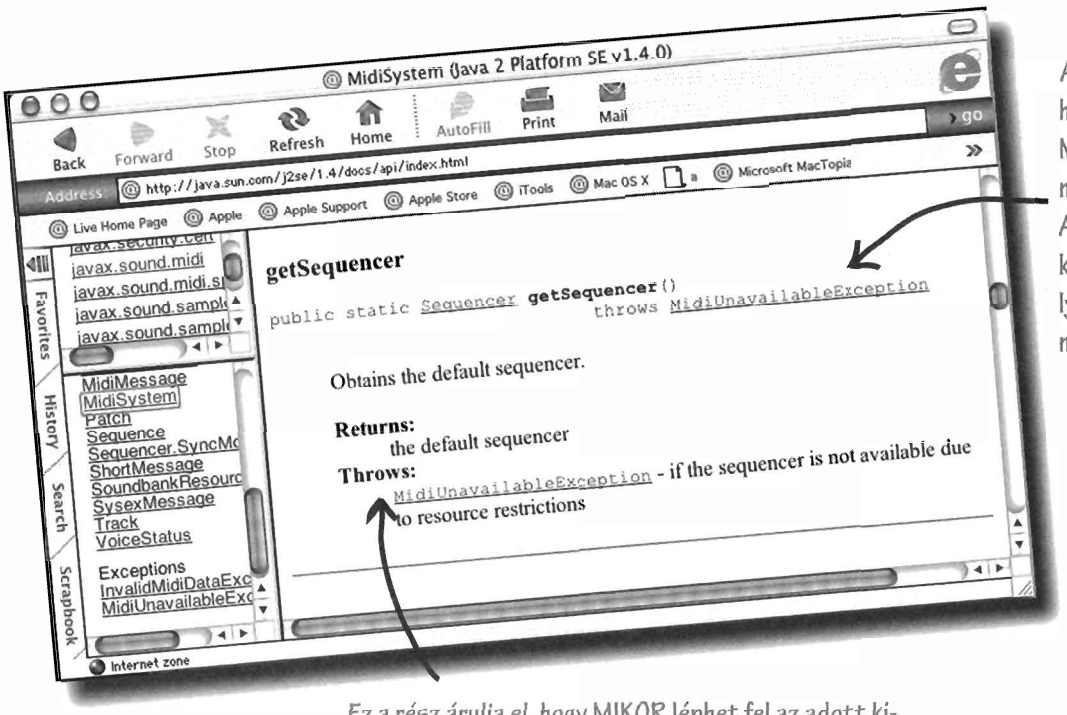


A tagfüggvények a Javában kivételek segítségével közlik a hívó kóddal, hogy

„Valami gond van. Nem sikerült végrehajtanom a feladatot."

A Java kivételkezelő rendszere tiszta, áttekinthető módot ad a futás közben esetleg felmerülő „kivételes helyzetek" kezelésére: lehetővé teszi, hogy egyetlen, könnyen elérhető helyre gyűjtsd az összes hibakezelő kódot. Az egész azon alapul, hogy *tudod*, hogy a tagfüggvény, amelyet meghívsz, kockázatos (vagyis *esetleg* kivételt válthat ki), így kódot írhatsz ennek az eshetőségnek a kezelésére. Ha *tudod*, hogy kivételt kaphatsz, amikor meghívsz egy bizonyos tagfüggvényt, *felkészülhetsz* a kivételt okozó probléma elhárítására – és talán *helyre is állíthatod* a program működését.
De *honnan* tudhatod, hogy egy tagfüggvényt kivételt válthat ki? Nos, onnan, hogy a kockázatos tagfüggvények bevezetésében szerepel egy **`throws`** („dob", kivált) záradék.

A `getSequencer()` tagfüggvény kockázatot vállal, és futáskor hibát okozhat, ezért be kell „jelentenie", hogy neked milyen kockázattal jár, ha meghívod.

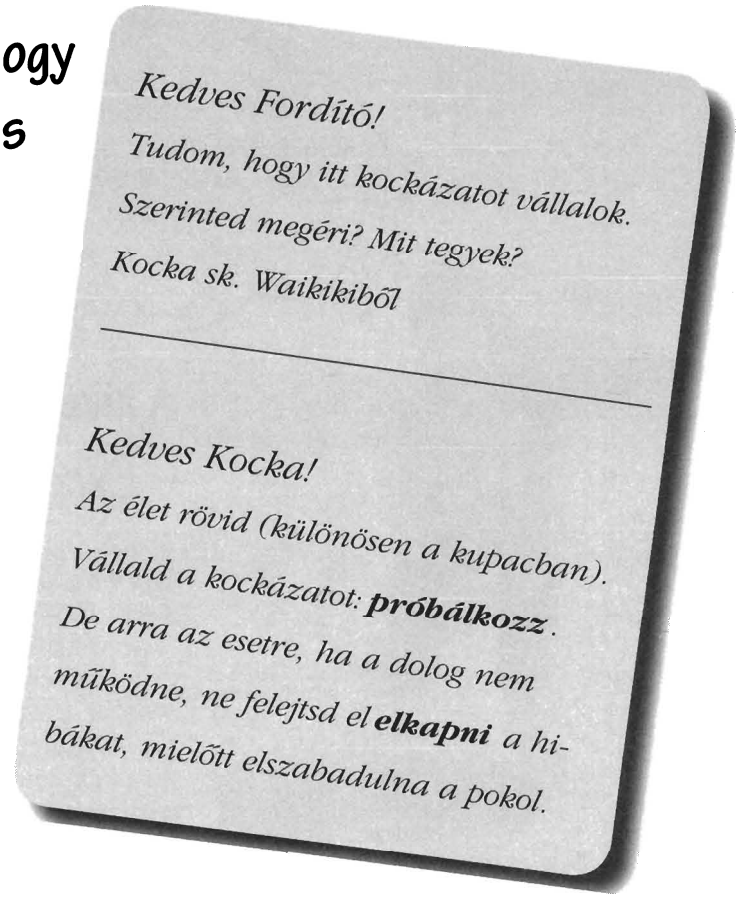


Az API elárulja neked, hogy a getSequencer() MidiUnavailableException kivételt válthat ki. A tagfüggvényeknek be kell jelenteniük, hogy milyen kivételeket válthatnak ki.

Ez a rész árulja el, hogy MIKOR léphet fel az adott kivétel – ebben az esetben az erőforrások korlátozottsága miatt (ami lehet, hogy csak annyit jelent, hogy a sorozat-feldolgozó már használatban van).

A fordítónak tudnia kell, hogy TE tudod, hogy kockázatos tagfüggvényt hívsz meg.

Ha a kockázatos kódot egy úgynevezett **try/catch** blokkba burkolod, a fordító megnyugszik.
A try/catch (próba/elkapás) blokk közli a fordítóprogrammal, hogy *tudod*, hogy a hívott tagfüggvény kivételes helyzetet idézhet elő, de készen állsz a kezelésére. A fordítót nem érdekli, hogy *miként* kezeled a helyzetet, csak az, hogy közölted, hogy gondoskodni fogsz róla.




```
import javax.sound.midi.*;

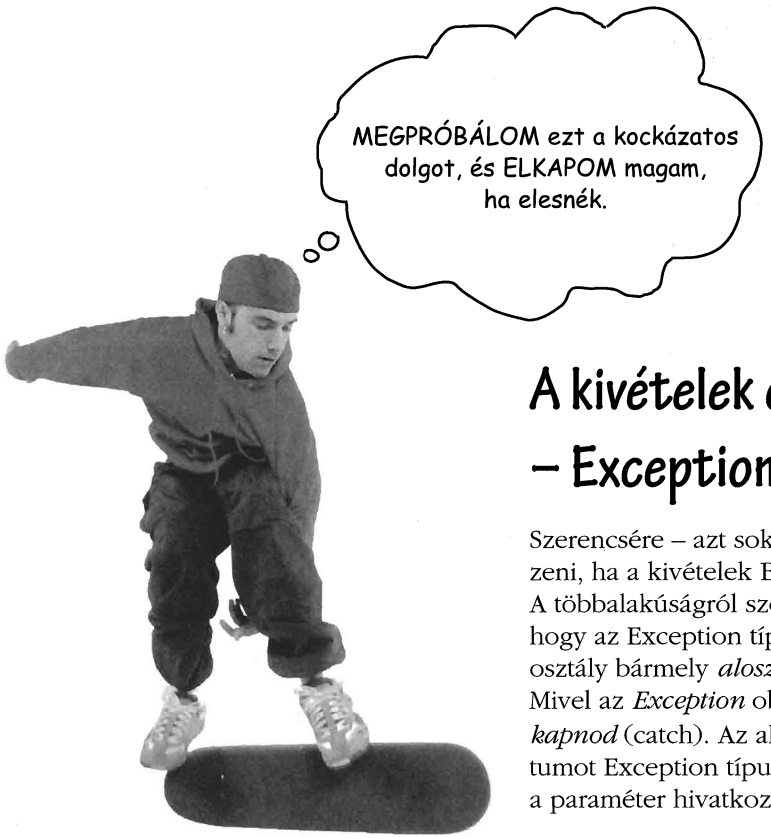
public class MusicTest1 {
    public void play() {

        try {
            Sequencer sequencer = MidiSystem.getSequencer();
            System.out.println("Successfully got a sequencer");
        } catch(MidiUnavailableException ex) {
                System.out.println("Bummer");
        }
    } // a play lezárása

    public static void main(String[] args) {
        MusicTest1 mt = new MusicTest1();
        mt.play();
    } // a main lezárása
} // az osztály lezárása
```

A kockázatos dolgot tedd egy „try" blokkba.

Egy „catch" blokkban írd le, mit kell tenni, ha a kivételes helyzet bekövetkezik, vagyis ha a getSequencer() hívása MidiUnavailableException kivételt vált ki.

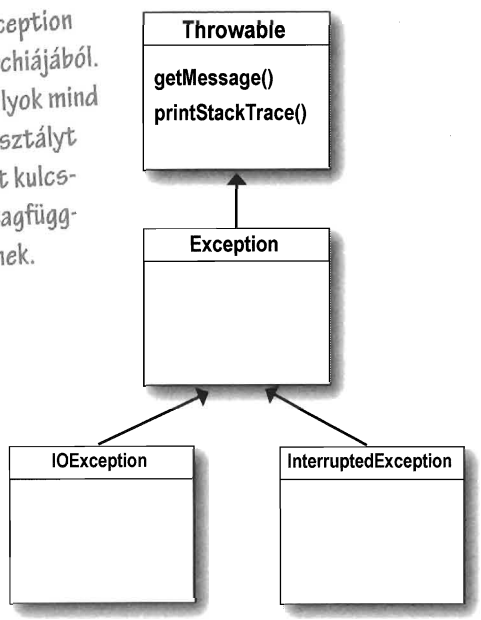


Ezt ne próbáld ki otthon.

A kivételek objektumok – Exception (Kivétel) típusúak

Szerencsére – azt sokkal nehezebb lenne megjegyzeni, ha a kivételek Brokkoli típusúak lennének. A többalakúságról szóló fejezetekből emlékezhetsz rá, hogy az Exception típusú objektumok az Exception osztály bármely *alosztályának* példányai is *lehetnek*. Mivel az *Exception* objektum, egy objektumot kell *elkapnod* (catch). Az alábbi kódban a **`catch`** argumentumot Exception típusúként vezetjük be, az *ex* pedig a paraméter hivatkozó változója.

Részlet az Exception osztály hierarchiájából. Ezek az osztályok mind a Throwable osztályt bővítik, és két kulcsfontosságú tagfüggvényt örökölnek.



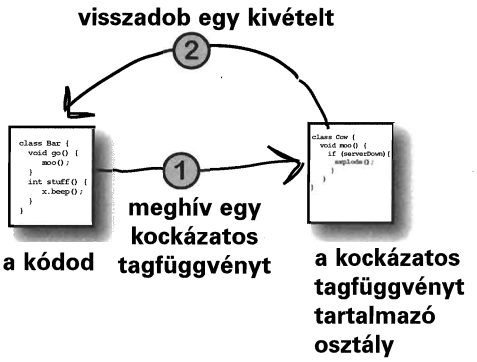

```
try {
  // kockázatos művelet
} catch(Exception ex) {
  // kísérlet a helyreállításra
}
```

ez pont olyan, mint egy tagfüggvény-argumentum bevezetése

Ez a kód csak akkor fut le, ha kivétel lép fel.

Az, hogy mit írsz a catch blokkba, a kiváltott kivételtől függ. Például ha a kiszolgáló nem működik, a catch blokkban esetleg próbálkozhatsz egy másik kiszolgálóval, ha pedig a fájl nem található, megkérheted a felhasználót, hogy segítsen megkeresni.

Ha az *én* kódom *kapja el* a kivételt, akkor kinek a kódja *váltja ki*?



Amikor Java-kódot írsz, sokkal több időt fogsz a kivételek *kezelésével* tölteni, mint azzal, hogy magad *hozd létre* és *váltsd ki* őket. Egyelőre elég annyit tudnod, hogy amikor a kódod egy kockázatos tagfüggvényt *hív meg* – tehát olyan tagfüggvényt, amely kivételt vezet be – az említett tagfüggvény *dobja neked*, a hívónak a kivételt.

A valóságban előfordul, hogy te írod meg mindkét osztályt. Nem számít, ki írja a kódot, csak az, hogy tudd, melyik tagfüggvény *váltja ki* („dobja") a kivételt, és melyik tagfüggvény *kapja el*.

Ha olyan kódot írsz, amely kivételt válthat ki, a kivételt *be kell jelentened*.

1 **Kockázatos, kivételt kiváltó kód:**

ennek a tagfüggvények közölnie KELL a világgal, hogy BadException kivételt vált ki

```
public void takeRisk() throws BadException {
   if (abandonAllHope) {
     throw new BadException();
   }
}
```

létrehozunk egy új Exception objektumot, és „odadobjuk"

2 **A kockázatos tagfüggvényt meghívó kód:**

```
public void crossFingers() {
   try {
     anObject.takeRisk();
   } catch (BadException ex) {
     System.out.println("Aaargh!");
     ex.printStackTrace();
}
```

Ha a programot nem tudod helyreállítani a kivétel után, LEGALÁBB kérj egy veremlenyomatot a printStackTrace() tagfüggvénnyel, amelyet minden kivétel örököl.

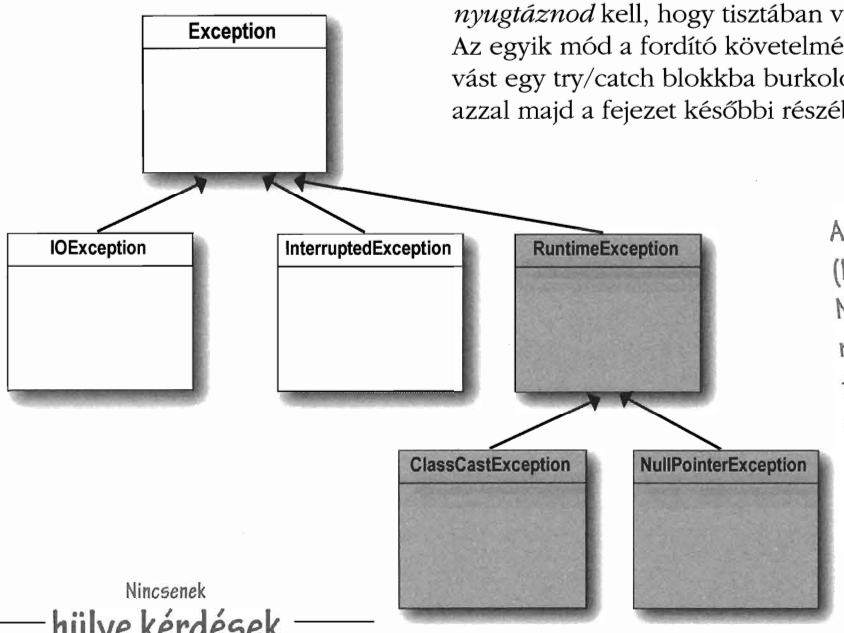
Amit egy tagfüggvény *dob*, azt egy másik tagfüggvény *elkapja*. A kivételeket mindig a hívónak dobják vissza.
A kivételt dobó tagfüggvénynek *be kell jelentenie*, hogy kivételt válthat ki.

A fordító mindent ellenőriz, kivéve a futásidejű kivételeket

Azokat a kivételeket, amelyek NEM alosztályai a RuntimeException-nek (FutásidejűKivétel), a fordító ellenőrzi, ezért „ellenőrzött kivételeknek" (checked exception) hívják őket.

A fordító követelményei:

1. Ha a kódodban kivételt *dobsz*, a tagfüggvény bevezetésekor be *kell* jelentened azt a *throws* kulcsszóval.
2. Ha olyan tagfüggvényt *hívsz meg*, amely kivételt dob (vagyis olyan tagfüggvényt, amely *bejelenti* – „deklarálja" –, hogy kivételt vált ki), *nyugtáznod* kell, hogy tisztában vagy a kivétel lehetőségével.
   Az egyik mód a fordító követelményének kielégítésére, hogy a hívást egy try/catch blokkba burkolod. (Van egy másik módszer is, de azzal majd a fejezet későbbi részében foglalkozunk.)



A futásidejű kivételeket (RuntimeException) a fordító NEM ellenőrzi, ezért ezek (minő meglepetés!) „ellenőrizetlen kivétel" (unchecked exception)) néven ismertek. Futásidejű kivételt is kiválthatsz, elkaphatsz és bevezethetsz, de nem muszáj, és a fordító nem fogja ellenőrizni őket.

Nincsenek

hülye kérdések

K: Várjunk csak egy percet! Hogy lehet, hogy ez az ELSŐ alkalom, amikor try/catch blokkal kell kezelnünk egy kivételt? Korábban már beleszaladtunk olyan kivételekbe, mint a NullPointerException, a DivideByZero kivétele, sőt az Integer.parseInt() tagfüggvénytől NumberFormatException kivételt is kaptunk. Azokat miért nem kellett elkapnunk?

V: A fordító az Exception valamennyi alosztályáról gondoskodik, *kivéve* ha azok RuntimeException típusúak. Minden kivételosztály, amely a RuntimeException-t bővíti, zöld utat kap. RuntimeException-t bárhol ki lehet váltani, throws bejelentéssel vagy try/catch blokkal, vagy ezek nélkül. A fordítót nem érdekli, hogy egy tagfüggvény bejelenti-e egy RuntimeException kiváltását, vagy hogy a hívó nyugtázza-e, hogy futáskor ilyen kivételt kaphat.

K: Eldobom az agyam. MIÉRT nem érdeklik a fordítót a futásidejű kivételek? Nem ugyanannyi az esélye, hogy térdre kényszerítik a programot?

V: A legtöbb futásidejű kivétel a kód logikájának hibájából ered, nem pedig egy olyan feltételből, ami futáskor esetleg megjósolhatatlan, illetve elkerülhetetlen módon nem teljesül. Azt *nem* garantálhatod, hogy egy fájl megtalálható-e a kívánt helyen, ahogy *azt sem*, hogy a kiszolgáló éppen működik-e, arról viszont *igenis* gondoskodhatsz, hogy a kódod ne szaladjon túl egy tömb végén (ezért van a length jellemző).

A futásidejű kivételekkel fejlesztés és tesztelés közben KELL találkoznod. Biztosan nem szeretnéd például egy olyan try/catch blokkban elhelyezett kóddal terhelned a programod, ami valami olyasmit kap el, aminek nem is lenne szabad bekövetkeznie.

A try/catch blokkok a kivételes helyzetek, és nem a kódodban levő hibák kezelésére valók. A catch blokkot arra használd, hogy megkíséreld helyreállítani a programot egy olyan művelet után, aminek a sikerét nem tudod garantálni, vagy legalábbis kiírj egy üzenetet és egy veremlenyomatot a felhasználónak, hogy kitalálhassa, mi történt.

Újratöltve



- A tagfüggvények kivételt válthatnak ki, ha futáskor valami nem sikerül.
- A kivételek mindig Exception típusú objektumok. (Ahogy a többalakúságról szóló fejezetekből emlékezhetsz rá, ez azt jelenti, hogy az objektum egy olyan osztályból származik, amelynek az öröklési fájában valahol feljebb megtalálható az Exception osztály.)
- A fordító NEM foglalkozik a **RuntimeException** típusú kivételekkel. A RuntimeException kivételeket nem kell bejelenteni, illetve try/catch blokkba burkolni (bár mindkettőt megteheted).
- Azokat a kivételeket, amelyek érdeklik a fordítót, „ellenőrzött kivételeknek" hívják, ami azt jelenti, hogy a *fordító* ellenőrzi őket. A fordító csak a RuntimeException kivételeket zárja ki az ellenőrzésből; a szabályoknak megfelelően minden más kivételt nyugtáznod kell a kódban.
- Egy tagfüggvény a `throw` kulcsszóval válthat ki kivételt, amelyet egy új kivételobjektum követ:

```
throw new NoCaffeineException();
```

- Azoknak a tagfüggvényeknek, amelyek *esetleg* ellenőrzött kivételt válthatnak ki, ***kötelező*** bejelenteniük ezt egy `throws Exception` deklarációval.
- Ha a kódod ellenőrzött kivételt kiváltó tagfüggvényt hív meg, biztosítania kell a fordítót, hogy megtette a szükséges óvintézkedéseket.
- Ha kezelni szeretnél egy kivételt, burkold a hívást egy try/catch blokkba, és tedd a kivételkezelő/helyreállító kódot a catch blokkba.
- Ha nem állsz készen a kivétel kezelésére, akkor is a fordító kedvére tehetsz, ha hivatalosan „lebuksz" a kivétel elől. A „lebukásról" (ducking) egy kicsit később lesz szó.

Metakognitív tipp

Amikor valami újat próbálsz megtanulni, mindig az legyen az *utolsó*, aminek a megjegyzésével próbálkozol, mielőtt lefekszel aludni. Tehát miután letetted ezt a könyvet (már ha el tudsz szakadni tőle), ne olvass semmit, ami nagyobb kihívást jelent egy Cheerios™-doboz hátuljánál. Az agyadnak időre van szüksége ahhoz, hogy feldolgozza, amit olvastál és tanultál, és ez beletelhet néhány *órába*. Ha rögtön rálapátolsz valami újat a Java-ismeretekre, lehet, hogy egyes dolgok nem ragadnak meg belőle. Ez a szabály természetesen nem zárja ki a testedzést. Nyugodtan végezz kickbox-gyakorlatokat – azok valószínűleg nem befolyásolják a Java-tanulást. A legjobb eredményt úgy érheted el, ha közvetlenül elalvás előtt olvasgatod ezt a könyvet (vagy legalábbis nézegeted benne a képeket).



Hegyezd ki a ceruzád!



Az itt szereplő műveletek közül szerinted melyek válthatnak ki olyan kivételt, ami érdekli a fordítót? Csak olyasmit keresünk, ami felett nincs hatalmad a kódodban. Az első sort kitöltöttük helyetted.
(Mert az volt a legkönnyebb.)

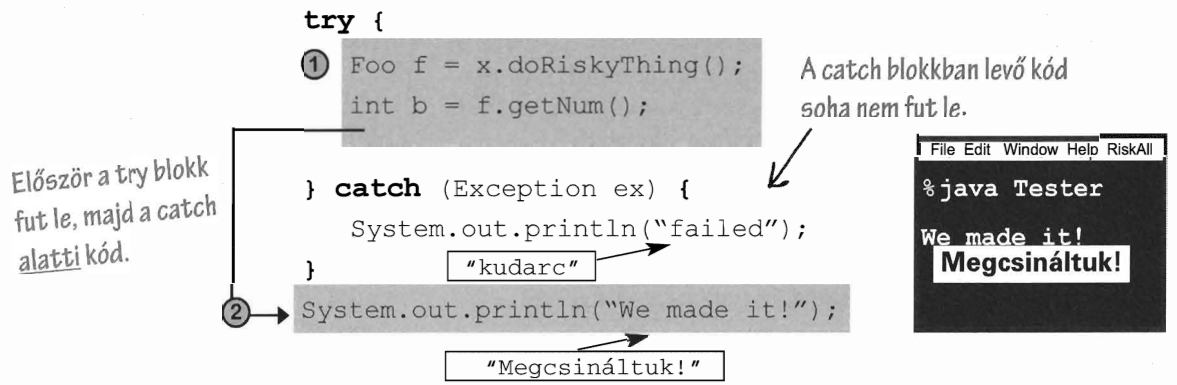
Amit csinálni akarsz:	Ami gondot okozhat:
kapcsolódni egy távoli kiszolgálóhoz	*a kiszolgáló nem működik*
túlhaladni egy tömb hosszán	
megjeleníteni egy ablakot a képernyőn	
adatokat kinyerni egy adatbázisból	
megvizsgálni, hogy egy szövegfájl ott van-e, ahol *gondolod*	
új fájlt létrehozni	
kiolvasni egy karaktert a parancssorból	

Folyamatvezérlés a try/catch blokkokban

Amikor kockázatos tagfüggvényt hívsz meg, két dolog történhet: a kockázatos tagfüggvény vagy sikerrel jár, és a try blokk befejeződik, vagy a kockázatos tagfüggvény kivételt dob vissza a hívó tagfüggvénynek.

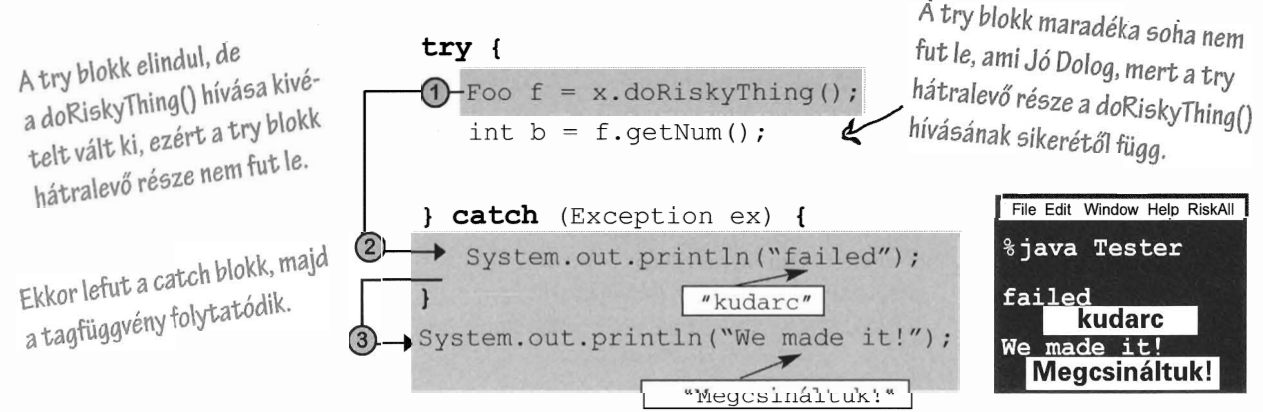
Ha a try *sikerrel jár*

(a doRiskyThing() *nem* vált ki kivételt)



Ha a try *kudarcot vall*

(mert a doRiskyThing() *mégis* kivételt dob)



Finally: amit *végre* akarsz hajtani, ha törik, ha szakad

Ha sütni szeretnél, először bekapcsolod a sütőt.
Ha az étel, amivel próbálkozol, **tökéletes kudarc**, ***ki kell kapcsolnod a sütőt.***
Ha az étel, amivel próbálkozol, remekül **sikerül**, ***akkor is ki kell kapcsolnod a sütőt.***
A sütőt mindenképpen ki kell kapcsolnod!

Azt a kódot, amelynek minden kivételtől függetlenül le kell futnia, egy finally („végül") blokkba kell tenned.

```
try {
  turnOvenOn();
  x.bake();
} catch (BakingException ex) {
    ex.printStackTrace();
} finally {
    turnOvenOff();
}
```

A finally nélkül a turnOvenOff() – kikapcsolSuto() – tagfüggvényt a try *és* a catch blokkban *is* el kellene helyezned, mert a ***sütőt mindenképpen ki kell kapcsolnod***. A finally blokk lehetővé teszi, hogy minden fontos rendrakó kódot *egy* helyre tegyél, ilyen ismétlés nélkül:

```
try {
  turnOvenOn();
  x.bake();
  turnOvenOff();
} catch (BakingException ex) {
    ex.printStackTrace();
    turnOvenOff();
}
```



Ha a try blokk kudarcot vall (kivétel lép fel), a vezérlés azonnal a catch blokkhoz kerül. Amikor a catch blokk befejeződött, a finally blokk fut le, amikor pedig az is végzett, a tagfüggvény hátralevő része folytatódik.

Ha a try blokk sikerrel jár (*nem* lép fel kivétel), a vezérlés átugorja a catch blokkot, és a finally blokkhoz kerül. Ha a finally blokk végzett, a tagfüggvény hátralevő része folytatódik.

Ha a try vagy a catch blokkban return utasítás szerepel, a finally blokk akkor is lefut! A vezérlés a finally blokkra ugrik, majd vissza a return utasításhoz.

Hegyezd ki a ceruzád!

Folyamatvezérlés

Nézd meg a bal oldalon látható kódot. Mit gondolsz, milyen kimenetet eredményez ez a program? És mi lenne a kimenet, ha a program harmadik sora erre változna: `String test = "yes";`? (Feltételezzük, hogy a ScaryException az Exception bővítése.)

```
public class TestExceptions {

  public static void main(String [] args) {

    String test = "no";
    try {
       System.out.println("start try");
       doRisky(test);
       System.out.println("end try");
    } catch ( ScaryException se) {
        System.out.println("scary exception");
    } finally {
        System.out.println("finally");
    }
    System.out.println("end of main");
  }

  static void doRisky(String test) throws ScaryException {
    System.out.println("start risky");
    if ("yes".equals(test)) {
      throw new ScaryException();
    }
    System.out.println("end risky");
    return;
  }
}
```

Kimenet, ha `test = "no"`

Kimenet, ha `test = "yes"`

Ha test="no": try indít – risky indít – risky befejez – try befejez – finally – main vége
Ha test="yes": try indít – risky indít – scaryexception – finally – main vége

Említettük, hogy egy tagfüggvény több kivételt is dobhat?

Egy tagfüggvény több kivételt is kiválthat, ha nagyon muszáj neki. A tagfüggvény bevezetésében azonban be kell jelenteni az *összes* ellenőrzött kivételt, amit a tagfüggvény dobhat (bár ha két vagy több kivételnek közös alaposztálya van, elég, ha a tagfüggvény csak az alaposztályt jelenti be).

Több kivétel elkapása

A fordító meggyőződik róla, hogy *minden* ellenőrzött kivételt kezeltél, amit a meghívott tagfüggvény kiválthat. A *catch* blokkokat a *try* alatt kell egymásra pakolnod. Előfordulhat, hogy a catch blokkok sorrendje számít, de erről majd kicsit később.





```
public class Szennyes {
    public void kimosSzennyes() throws NadragKivetel, MelltartoKivetel {
    // kód, amely a két kivétel bármelyikét kiválthatja
    }
}
```

Ez a tagfüggvény kettő – számold meg –, azaz KETTŐ kivételt jelent be.

```
public class Foo {
    public void go() {
          Szennyes szennyes = new Szennyes();
          try {
             szennyes.kimosSzennyes();
          } catch(NadragKivetel nex) {
                // helyreállító kód
          } catch(MelltartoKivetel mex) {
                // helyreállító kód
          }
    }
}
```



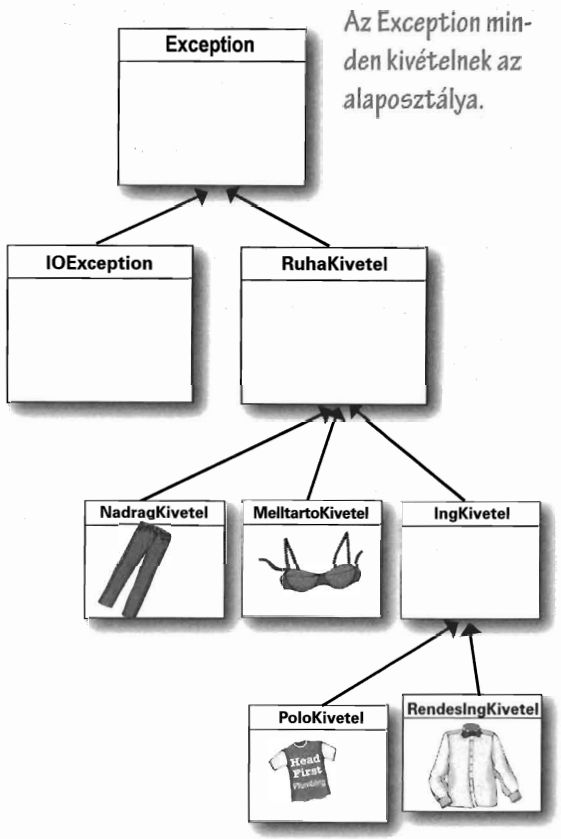
Ha a kimosSzennyes() NadragKivetel-t vált ki, akkor a NadragKivetel catch blokkjában találja magát.



Ha a kimosSzennyes() MelltartoKivetel-t vált ki, akkor a MelltartoKivetel catch blokkjára ugrik.

A kivételek többalakúak

A kivételek objektumok, emlékszel? Semmi különös nincs bennük, attól eltekintve, hogy ***dobni*** *lehet őket.* Így aztán mint minden tisztességes objektumra, a kivételekre is lehet többalakúan hivatkozni. Egy MelltartoKivetel *objektumot* például hozzá lehet rendelni egy RuhaKivetel *hivatkozáshoz*, egy NadragKivetel-t pedig egy Exception hivatkozáshoz. Biztos érted a lényeget.
A kivételek előnye, hogy a tagfüggvényeknek nem kell kifejezetten bejelenteniük minden lehetséges kivételt, amit kiválthatnak – elég, ha a kivételek alaposztályát jelentik be. Ugyanez érvényes a catch blokkokra is: nem kell minden lehetséges kivételhez catch blokkot írnod, amíg a meglevő catch blokk (vagy blokkok) képesek bármilyen nekik dobott kivételt kezelni.



① A kivételeket BEJELENTHETED a kiváltott kivételek alaptípusával is.



```
public void kimosSzennyes() throws RuhaKivetel{
```

A RuhaKivetel bevezetése a RuhaKivetel bármely alosztályának dobását lehetővé teszi. Ez azt jelenti, hogy a kimosSzennyes() NadragKivetel-t, MelltartoKivetel-t, PoloKivetel-t és RendesIngKivetel-t is kiválthat, anélkül, hogy kifejezetten bejelentené mindegyiket.

② A kivételeket ELKAPHATOD a kiváltott kivétel egy alaptípusával is.

```
try {
    szennyes.kimosSzennyes();
```



a RuhaKivetel bármelyik alosztályát el tudja kapni

```
} catch(RuhaKivetel, rex) {
    // helyreállító kód
}
```

```
try {
    szennyes.kimosSzennyes();
```

csak PoloKivetel-t és RendesIngKivetel-t tud elkapni



```
} catch(IngKivetel, iex) {
    // helyreállító kód
}
```

Az, hogy el TUDSZ kapni mindent egyetlen nagy szuper-többalakú catch blokkal, még nem jelenti azt, hogy mindig ÉRDEMES.

Megteheted, hogy úgy írod meg a kivételkezelő kódodat, hogy csak *egy* catch blokkot adsz meg, annak minden záradékában az Exception alaptípust használva, hogy *bármilyen* kivételt elkaphass, ami felléphet:

```
try {
    szennyes.kimosSzennyes();
} catch(Exception, ex) {
          // helyreállító kód...
}
```

Helyreállítás MIBŐL? Ez a catch blokk MINDEN kivételt elkap, így nem tudhatod automatikusan, hogy mi volt a gond.

Írj külön-külön catch blokkot minden olyan kivétel számára, amelyet egyedi módon kell kezelned.

Ha kódod például másképpen kezeli a PoloKivetel-eket (illetve máshogy állítja helyre a programot utánuk), mint ahogy a MelltartoKivetelek-eket, írj külön-külön catch blokkokat a számukra. Ha viszont a RuhaKivetel minden más típusát ugyanúgy kezeled, akkor adj egy RuhaKivetel-elkapó záradékot is a blokkhoz, hogy az kezelje a többi kivételt:

```
try {
    szennyes.kimosSzennyes();

} catch(PoloKivetel, pex) {
    // helyreállítás a PoloKivetel-ből

} catch(MelltartoKivetel, mex) {
    // helyreállítás a MelltartoKivetel-ből

} catch(RuhaKivetel, rex) {
    // helyreállítás minden másból
}
```







A PoloKivetel-ek és a MelltartoKivetel-ek más-más helyreállító kódot igényelnek, ezért különböző catch blokkokat kell használnod.

Minden más RuhaKivetel-t itt kapunk el.

A catch blokkokat a szűkebbtől a bővebb felé haladó rendben kell megadnod



Itt csak a PoloKivetel-eket kapod el, a többi nem fér bele.

```
catch(PoloKivetel pex)
```



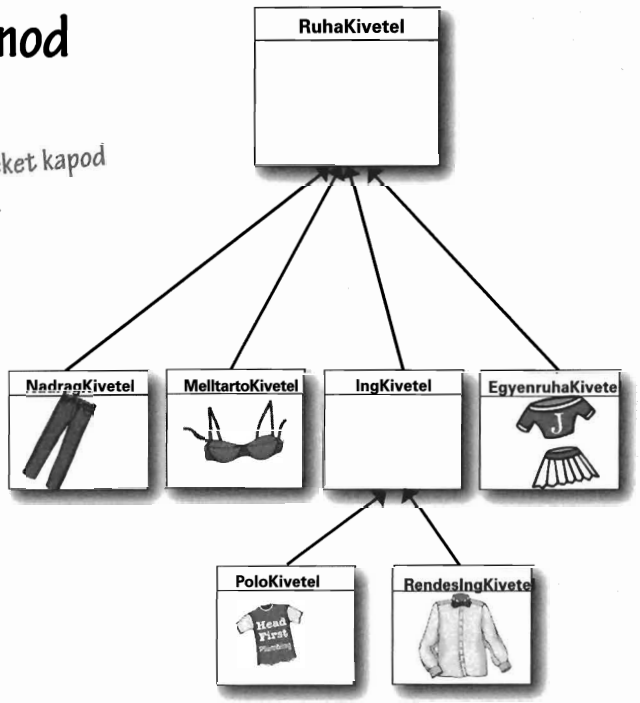
A PoloKivetel-ek soha nem jutnak el idáig, az összes többi IngKivetel-t viszont itt kapod el.

```
catch(IngKivetel iex)
```



Minden más RuhaKivetel-t itt kapsz el, a PoloKivetel-ek és IngKivetelek-ek viszont soha nem jutnak el idáig.

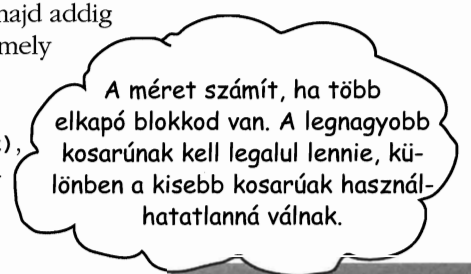
```
catch(RuhaKivetel rex)
```



Minél feljebb vagy az öröklési fában, annál nagyobb az elkapó „kosár", míg ha lefelé haladsz a fában, az egyre szűkebb és szűkebb körre kiterjedő Exception-osztályok felé, a kosár egyre kisebb lesz. Ez csupán egyszerű többalakúság. Az IngKivetel elkapó kosara elég nagy ahhoz, hogy beleférjen egy PoloKivetel vagy egy RendesIngKivetel (és az IngKivetel-t bővítő minden további alosztály). A RuhaKivetel kosara még ennél is nagyobb (mivel a RuhaKivetel típussal több dologra lehet hivatkozni) – belefér egy RuhaKivetel típusú kivétel (nahát!), és a RuhaKivetel minden alosztálya is: a NadragKivetel, az EgyenruhaKivetel, a MelltartoKivetel és az IngKivetel. Minden catch-argumentum anyja az **Exception** típus: ennek a kosara *bármilyen* kivételt elkap, beleértve a futásidejű (ellenőrizetlen) kivételeket is, ezért tesztelésen kívül valószínűleg nem fogod másra használni.

Nagyobb kosarat nem tehetsz egy kisebb kosár fölé

Illetve *megteheted*, de a fordítás nem fog sikerülni. A catch blokkok nem olyanok, mint a túlterhelt tagfüggvények, ahol a Java a legjobban illeszkedőt választja ki. A catch blokkok feldolgozását a JVM egyszerűen az elsővel kezdi, majd addig halad lefelé, amíg olyan catch blokkot nem talál, amely elég bő (más szavakkal: elég magasan található az öröklési fában) ahhoz, hogy kezelje a kivételt. Ha az első catch blokkod a `catch(Exception ex)`, akkor a fordító tudja, hogy nincs értelme továbbiakat hozzáadni, mert a vezérlés úgysem juthat el soha hozzájuk.




```
try{
  szennyes.kimosSzennyes();
```

Ezt ne csináld!




```
} catch(RuhaKivetel rex) {
  // helyreállítás a RuhaKivetel-ből
```



```
} catch(MelltartoKivetel mex) {
  // helyreállítás a MelltartoKivetel-ből
```



```
} catch(IngKivetel iex) {
  // helyreállítás az IngKivetel-ből
}
```



A testvérek bármilyen sorrendben állhatnak, mivel nem kaphatják el egymás kivételeit.

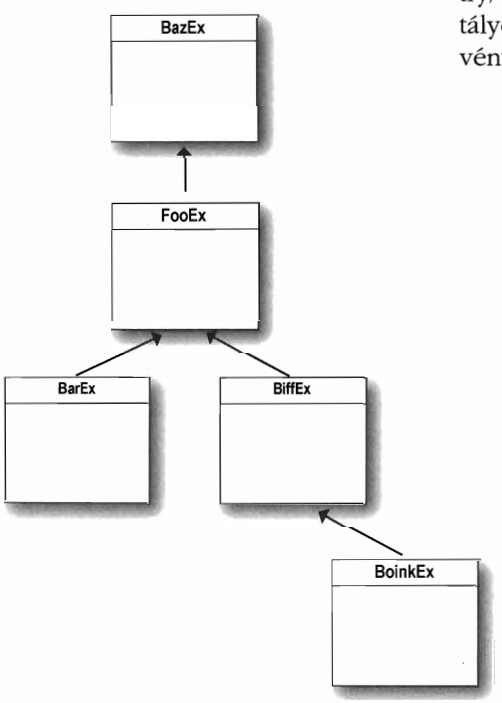
Az IngKivetel-t a MelltartoKivetel fölé helyezheted, senki nem fogja bánni. Bár az IngKivetel nagyobb (bővebb) típus, hiszen más osztályokat is el tud kapni (a saját alosztályait), a MelltartoKivetelek-et nem, úgyhogy nincs probléma.

Hegyezd ki a ceruzád!

Tételezzük fel, hogy az itt látható try/catch blokk kódja érvényes. A feladatod az, hogy felrajzolj két különböző osztálydiagramot, amelyek pontosan tükrözik a kivételosztályokat. Más szavakkal, milyen osztályöröklési szerkezetek esetén érvényes a példakód try/catch blokkja?

```
try {
  x.doRisky();
} catch(AlphaEx a) {
    // helyreállítás az AlphaEx-ből
} catch(BetaEx b) {
    // helyreállítás a BetaEx-ből
} catch(GammaEx c) {
    // helyreállítás a GammaEx-ből
} catch(DeltaEx d) {
    // helyreállítás a DeltaEx-ből
}
```

A feladatod az, hogy megírj két különböző (a fentihez hasonló) try/catch szerkezetet, amely pontosan megfelel a baloldalt látható osztálydiagramnak. Feltételezzük, hogy a try blokkot tartalmazó tagfüggvény MINDEGYIKET kiválthatja az itt szereplő kivételek közül.



Ha nem akarsz kezelni egy kivételt...

csak bukj le!



Ha egy kivételt nem akarsz kezelni, **bejelentheted**, hogy **lebuksz** előle.

Amikor kockázatos tagfüggvényt hívsz meg, a fordító elvárja, hogy nyugtázd a veszélyt. A legtöbbször ez azt jelenti, hogy a kockázatos hívást egy try/catch blokkba burkolod. Van azonban egy másik lehetőség is: egyszerűen *lebukhatsz*, és hagyhatod, hogy a *te kódodat* meghívó tagfüggvény kapja el a kivételt.

A dolog egyszerű: csak annyit kell tenned, hogy *bejelented*, hogy *te* dobod a kivételeket. Az nem számít, hogy valójában *nem te* hajtod végre a dobást, ugyanis te vagy az, aki hagyja a kivételt elsüvíteni.

Ha viszont lebuksz egy kivétel elől, akkor try/catch blokkra sincs szükséged. Mit történik hát, ha a kockázatos függvény – a kimosSzennyes() – *tényleg* kiváltja a kivételt?

Amikor egy tagfüggvény kivételt vált ki, az említett tagfüggvény azonnal lepattan a verem tetejéről, és a kivétel a verem következő tagfüggvényéhez kerül – a *hívóhoz*. Ha viszont a hívó *lebukik*, akkor nem kapja el a kivételt, így a *hívó* is rögtön lepattan a veremről, a kivétel pedig a következő tagfüggvényhez kerül, és így tovább. Hol ér véget a lánc? Mindjárt megtudod.

VALÓJÁBAN nem dobsz ilyen kivételt, de mivel nincs try/catch blokkod a meghívott kockázatos tagfüggvény számára, TE MAGAD válsz „kockázatos tagfüggvénnyé" – bárki is hívjon meg TÉGED, foglalkoznia kell a kivétellel.

```
public void foo() throws NagyonRosszKivetel {
   // kockázatos tagfüggvény meghívása try/catch nélkül
   szennyes.kimosSzennyes();
}
```

A lebukás bejelentése csak késlelteti az elkerülhetetlent

Előbb vagy utóbb valakinek kezelnie kell a kivételt. De mi történik, ha a main() bukik le a kivétel elől?

Mindkét tagfüggvény bejelenti, hogy lebukik a kivétel elől, így azt senki nem kezeli. Ez gond nélkül lefordítható.

```
public class Washer {
    Laundry laundry = new Laundry();

    public void foo() throws ClothingException {
        laundry.doLaundry();
    }

    public static void main (String[] args) throws ClothingException {
        Washer a = new Washer();
        a.foo();
    }
}
```

1 A kimosSzennyes() RuhaKivetel-t vált ki.



A main() meghívja a foo()-t.
A foo() meghívja a kimosSzennyes()-t.
A kimosSzennyes() lefut, és RuhaKivetelt-t vált ki.

2 A foo() lebukik a kivétel elől.



A kimosSzennyes() azonnal lepattan a veremről, és a kivétel visszakerül a foo()-hoz. A foo() azonban nem rendelkezik try/catch blokkal, ezért…

3 A main() lebukik a kivétel elől.



A foo() azonnal lepattan a veremről, és a kivétel visszakerül a… kihez is? Micsoda?! Senki sem maradt, csak a JVM, aki azt mondja: „TŐLEM aztán ne várd, hogy kihúzlak ebből!".

4 A JVM lekapcsol.

A RuhaKivetel-eket a pólóval ábrázoljuk. Persze, tudjuk… te inkább a farmert szeretted volna.

Kezeld vagy bukj le! Ez a törvény.

Most már láttad mind a két módszert arra, hogy kedvére tégy a fordítónak, amikor kockázatos (kivételt dobó) tagfüggvényt hívsz meg.

1 KEZELÉS

Burkold a kockázatos hívást egy try/catch blokkba.

```
try {
  szennyes.kimosSzennyes();
} catch(RuhaKivetel rex) {
  // helyreállító kód
}
```

Ennek elég nagy elkapónak kell lennie ahhoz, hogy az összes kivételt kezelje, amit a kimosSzennyes() kiválthat, különben a fordító panaszkodni fog, hogy nem kapsz el minden kivételt.

2 LEBUKÁS (bejelentve)

Jelentsd be, hogy a TE tagfüggvényed ugyanazokat a kivételeket dobja, mint a meghívott kockázatos tagfüggvény.

```
void foo() throws RuhaKivetel {
  szennyes.kimosSzennyes();
}
```

A kimosSzennyes() tagfüggvény RuhaKivetel-t vált ki, de a kivétel bejelentésével végül a foo() tagfüggvény bukik le a kivétel elől. Try/catch blokk nincs.

Ez azonban azt jelenti, hogy bárki is hívja meg a foo() tagfüggvényt, be kell tartania a Kezeld vagy bukj le! törvényt. Ha a foo() lebukik a kivétel elől, és a main() hívja meg a foo()-t, akkor a main()-nek kell gondoskodnia a kivételről.

```
public class Moso {
  Szennyes szennyes = new Szennyes();

  public void foo() throws RuhaKivetel {
    szennyes.kimosSzennyes();
  }

  public static void main (String[] args) {
    Moso a = new Moso();
    a.foo();
  }
}
```

BAJ VAN!!!
A main() most nem fordítható le, és „unreported exception" hibát kapsz. A fordító szempontjából a foo() tagfüggvény kivételt vált ki.

Mivel a foo() tagfüggvény lebukik a kimosSzennyes() által kiváltott RuhaKivetel elől, a main()-nek vagy egy try/catch blokkba kell burkolnia az a.foo()-t, vagy be kell jelentenie, hogy ő is RuhaKivetel-t vált ki.

Vissza a zenegép kódjához...

Biztosan sikerült teljesen elfeledtetni veled, hogy a fejezetet egy JavaSound-kódkezdeménnyel kezdtük. Létrehoztunk egy Sequencer objektumot, de nem sikerült lefordítani, mert a Midi.getSequencer() tagfüggvény ellenőrzött kivételt (MidiUnavailableException) jelentett be. Most már kijavíthatod a kódot: a hívást egy try/catch blokkba kell burkolnod:

```
public void play() {
        try {

            Sequencer sequencer = MidiSystem.getSequencer();
            System.out.println("Successfully got a sequencer");
        } catch(MidiUnavailableException ex) {
            System.out.println("Bummer");
        }
    } // a play lezárása
```

"Megvan a sequencer!"

"A fenébe."

Most, hogy egy try/catch blokkba burkoltuk, a getSequencer() nyugodtan meghívható.

A catch paraméterének a „megfelelő" kivételnek kell lennie. Ha azt írnád, hogy „catch(FileNotFoundException f)", akkor a kódot nem lehetne lefordítani, mert a többalakúság szabályai szerint egy MidiUnavailableException nem passzol bele egy FileNotFoundException kivételbe. Ne feledd: egy catch blokk önmagában nem elég – azt kell elkapnod, amit dobnak!

A kivételek szabályai

1 Nem lehet catch vagy finally blokkod try nélkül.

```
void go() {
   Foo f = new Foo();
   f.foof();
   catch(FooException ex) { }
}
```

NEM ÉRVÉNYES! Hol a try?

2 A try és a catch közé nem tehetsz kódot.

```
try {
   x.doStuff();
}
int y = 43;
} catch(Exception ex) { }
```

NEM ÉRVÉNYES! A try és a catch közé nem tehetsz kódot!

3 Egy try blokkot MINDIG egy catch vagy egy finally blokknak KELL követnie.

```
try {
   x.doStuff();
} finally {
   // rendrakás
}
```

ÉRVÉNYES, mert bár catch nincs, finally blokkal rendelkezel. A try viszont nem állhat önmagában.

4 Ha egy try blokkot csak egy finally blokk követ (catch nélkül), akkor is be kell jelenteni a kivételt.

```
void go() throws FooException {
    try {
       x.doStuff();
    } finally { }
}
```

Egy try catch nélkül nem tesz eleget a Kezeld vagy bukj le! törvénynek.

Kódkonyha



Nem kell mindent magadnak csinálnod, de úgy sokkal szórakoztatóbb.

A fejezet hátralevő része nem kötelező; a zeneprogramokhoz nyugodtan használhatsz készre sütött kódokat is. De ha többet szeretnél tudni a JavaSound-ról, lapozz tovább!

Valódi hangkeltés

Ha még emlékszel, a fejezet elején megtanultad, hogy a MIDI-adatok arra tartalmaznak utasításokat, hogy *mit* kell lejátszani (és *hogyan*), de ténylegesen *nem hoznak létre semmilyen hallható hangot.* Ahhoz, hogy hang jöjjön ki a hangszóróból, a MIDI-adatokat át kell vezetni valamilyen MIDI-eszközön, amely a MIDI-utasításokat hangokra képezi le, vagy egy hardveres, vagy egy „virtuális" hangszer (szoftveres szintetizátor) segítségével. Ebben a könyvben csak szoftveres eszközöket használunk, ezért az alábbiakban azt mutatjuk be, hogyan működik mindez a JavaSound-ban:

NÉGY dologra van szükség:

1. **Valamire, ami lejátssza a zenét.**
2. **A lejátszandó zenére – vagyis egy dalra.**
3. **Az utasítássorozatnak arra a részére, amely a tényleges információt tárolja.**
4. **A konkrét zenei információra: milyen hangokat kell lejátszani, meddig kell kitartani azokat stb.**

Sorozat-feldolgozó lejátssza **Utasítássorozat** tartalmazza **Sáv** tárolja

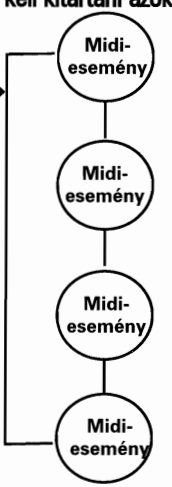


A Sorozat-feldolgozó (sequencer) az, ami lejátssza a dalt. Úgy gondolhatsz rá, mint egy **CD-lejátszóra**.



Az Utasítássorozat (sequence) a dal, a zeneszám, amelyet a sorozat-feldolgozó lejátszik. Ebben a könyvben úgy gondolhatsz rá, mint egy zenei CD-re, **amelyen azonban csak egy dal található**.

Ebben a könyvben csak egy számra van szükségünk, ezért egy egyetlen dalt tartalmazó – egyszámos, egy sávból álló – zenei CD-t képzelj el. Ez a Sáv (track) az, ahol a dal összes adata (MIDI-információja) található.



A MIDI-események (Midi event) olyan üzenetek, amelyeket a Sorozat-feldolgozó megért. Egy MIDI-esemény mondhatja például azt (ha tud angolul), hogy „ekkor és ekkor játszd le a középső C-t, ilyen gyorsan és ilyen hangosan, és tartsd ki ennyi és ennyi ideig", vagy azt, hogy „a hangszert változtasd most fuvolára".

Ebben a könyvben az Utasítássorozatra gondolj úgy, mintha egy egyetlen dalt tartalmazó (egy sávból álló) CD lenne. A dal lejátszásának módjáról a Sáv ad információt, a Sáv pedig az Utasítássorozat része.

És ÖT lépést kell végrehajtanod:

1. Létre kell hoznod egy **Sorozat-feldolgozót**, és meg kell nyitnod:

```
Sequencer player = MidiSystem.getSequencer();
player.open();
```

2. Létre kell hoznod egy új **Utasítássorozatot**:

```
Sequence seq = new Sequence(timing,4);
```

3. Létre kell hoznod egy új **Sávot** az Utasítássorozatból:

```
Track t = seq.createTrack();
```

4. Fel kell töltened a Sávot **Midi-eseményekkel**, és át kell adnod az Utasítássorozatot a Sorozat-feldolgozónak:

```
t.add(myMidiEvent1);
player.setSequence(seq);
```



Az első zenelejátszó programod

Írd be az alábbi kódot, és futtasd le – olyasmit fogsz hallani, mintha valaki leütött volna egy hangot a zongorán. (Na jó, nem vala*ki*, hanem vala*mi*.)

```
import javax.sound.midi.*;
```
← Ne felejtsd el betölteni a midi csomagot.

```
public class MiniMiniMusicApp {

    public static void main(String[] args) {
       MiniMiniMusicApp mini = new MiniMiniMusicApp();
       mini.play();
    } // a main lezárása

  public void play() {

    try {

      1  Sequencer player = MidiSystem.getSequencer();
         player.open();

      2  Sequence seq = new Sequence(Sequence.PPQ, 4);

      3  Track track = seq.createTrack();

      4  ShortMessage a = new ShortMessage();
         a.setMessage(144, 1, 44, 100);
         MidiEvent noteOn = new MidiEvent(a, 1);
         track.add(noteOn);

         ShortMessage b = new ShortMessage();
         b.setMessage(128, 1, 44, 100);
         MidiEvent noteOff = new MidiEvent(b, 16);
         track.add(noteOff);

         player.setSequence(seq);

         player.start();

    } catch (Exception ex) {
        ex.printStackTrace();
    }
  } // a play lezárása
} // az osztály lezárása
```

Létrehozunk egy Sequencer-t, és megnyitjuk (másképp nem használhatjuk – a Sequencer nem megnyitva érkezik).

A Sequence konstruktor argumentumaival ne törődj, csak másold le ezeket (mintha készre sütött argumentumok lennének).

Kérünk egy Track-et a Sequence-től. Ne feledd: a Sáv (Track) az Utasítássorozatban (Sequence) található, a MIDI-adatok pedig a Sávban.



A Track-ben elhelyezünk néhány Midi-eseményt (MidiEvent). Ez a rész nagyjából készre sütött kód. Csak a setMessage() tagfüggvény és a MidiEvent konstruktor argumentumaival kell foglalkoznod – ezeket a következő oldalon tárgyaljuk.

A Sequencer-nek átadjuk a Sequence-t (mintha egy CD-t tennénk a CD-lejátszóba).

Elindítjuk – start() – a Sequencer-t (ez olyan, mintha megnyomnánk a PLAY gombot).

Midi-esemény (dal-adatok) létrehozása

A Midi-események (MidiEvent) olyan utasítások, amelyek a dal egy-egy részletét határozzák meg. A Midi-események sorozata egyfajta kotta, vagy olyan, mint a gépzongora lyukkártyaszerű hengere. A számunkra érdekes Midi-események többsége ***egy lejátszandó dolgot*** ír le, illetve azt, hogy azt a bizonyos dolgot ***mikor kell lejátszani***. Az idő lényeges, mivel a zenében minden időzítés kérdése: ez a hang ezt és ezt a hangot követi, és így tovább. Mivel a Midi-események igen részletesek, meg kell mondanod, hogy a hang lejátszását mikor kell *megkezdeni* (noteOn esemény), és mikor kell *abbahagyni* (noteOff esemény). Ebből kitalálhatod, hogy a „hagyd abba a G hang lejátszását" (noteOff) nem lehet a „kezdd el a G hang lejátszását" (noteOn) üzenet *előtt*.
A MIDI-utasítások egy-egy Message (Üzenet) objektumba kerülnek:
a MidiEvent az üzenetet párosítja az üzenetben meghatározott művelet végrehajtásának idejével. Más szavakkal, a Message olyasmit mond, hogy „kezdd el a középső C lejátszását", míg a MidiEvent valami olyasmit, hogy „hajtsd végre ennek az üzenetnek az utasítását a 4. ütemben".
Egy Message és egy MidiEvent mindig szükséges. Az Üzenet mondja meg, *mit* kell tenni, a Midi-esemény pedig azt, hogy *mikor*.

A Midi-események határozzák meg, hogy mit és mikor kell tenni. Minden utasításnak tartalmaznia kell az utasítás időzítését is, vagyis azt, hogy az adott dolognak melyik ütemre kell bekövetkeznie.

1. **Üzenet** létrehozása

```
ShortMessage a = new ShortMessage();
```

2. Az **Utasítás** elhelyezése az Üzenetben

```
a. setMessage(144, 1, 44, 100);
```

Ez az üzenet azt mondja, hogy „kezdd el lejátszani a 44-es hangot" (a többi számot a következő oldalon tárgyaljuk).

3. Új **Midi-esemény** létrehozása az Üzenettel

```
MidiEvent noteOn = new MidiEvent(a, 1);
```

Az utasítások az üzenetben találhatók, de a Midi-esemény adja meg azok végrehajtásának idejét. Ez a Midi-esemény azt mondja, hogy az „a" üzenet utasításait az első ütemben (1) kell végrehajtani.

4. A Midi-esemény hozzáadása a **Sávhoz**

```
track.add(noteOn);
```

Minden MidiEvent obejktumot egy Sáv (Track) tárol. Az Utasítássorozat (Sequence) rendezi őket annak megfelelően, hogy az egyes eseményeknek mikor kell bekövetkezniük, a Sorozat-feldolgozó (Sequencer) pedig ebben a sorrendben játssza le őket. Pontosan ugyanabban az időben sok-sok esemény is bekövetkezhet, például ha egyszerre két hangot akarsz lejátszani, vagy különböző hangszerekkel egyidejűleg különböző hangokat szeretnél megszólaltatni.

MIDI-üzenetek: a Midi-események szíve

Az eseménynek azt a részét, amelyik megmondja, *mit* kell tenni, egy MIDI-üzenet tárolja. Ez a tényleges utasítás, amelyet a sorozat-feldolgozóval végre szeretnél hajtatni. Az utasítás első argumentuma mindig az üzenet típusa. A másik három argumentumban átadott értékek az üzenet típusától függnek. A 144-es típusú üzenetek jelentése például az, hogy „HANG BE" (Note On). Egy „HANG BE" utasítás végrehajtásához azonban a sorozat-feldolgozónak tudnia kell pár dolgot. Ha csak ezt az utasítást kapná, a sorozat-feldolgozó azt mondaná: „Jó, játsszak le egy hangot, de *melyik csatornán*? Más szóval, dobhangot vagy zongorahangot akarsz? És *melyik* hangot? A középső C-t? Vagy egy DISZ-t? És ha már itt tartunk: milyen *sebességgel* játsszam le?". Egy MIDI-üzenet összeállításához létre kell hoznod egy ShortMessage- (RövidÜzenet) példányt, és meg kell hívnod a setMessage() (beállítÜzenet) tagfüggvényt, átadva neki az üzenet négy argumentumát. Ne feledd: az üzenet csak azt mondja meg, hogy *mit* kell tenni, ezért az üzenetet bele kell gyömöszölnöd egy eseménybe, amely leírja, hogy az üzenet utasításait *mikor* kell végrehajtani.

Az üzenetek felépítése

A setMessage() első argumentuma mindig az üzenet „típusát" jelképezi, míg a másik három argumentum az üzenet típusától függő különböző dolgokat.

```
a.setMessage(144,  1,  44,  100);
```
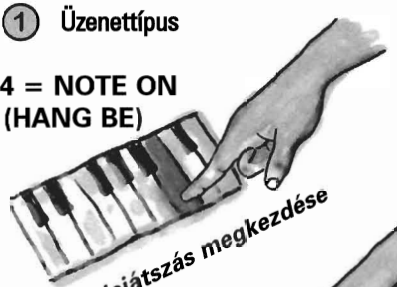
üzenettípus, csatorna, lejátszandóhang, sebesség

Az utolsó 3 argumentum az üzenet típusától függően más és más lehet. Ez egy HANG BE üzenet, ezért ezek az argumentumok azokat a dolgokat adják meg, amelyekre a sorozat-feldolgozónak szüksége van, hogy lejátszhassa a hangot.

Az *Üzenet* azt adja meg, hogy mit kell tenni, a *Midi-esemény* pedig azt, hogy mikor.

1 Üzenettípus

144 = NOTE ON (HANG BE)



a lejátszás megkezdése

128 = NOTE OFF (HANG KI)



a lejátszás befejezése

2 Csatorna

A csatornákra gondolj úgy, mintha zenészek lennének egy zenekarban. Az 1. csatorna (Channel 1) az 1. zenész (a billentyűs), a 9. csatorna a dobos, és így tovább.

3 Lejátszandó hang

Ez a szám adja meg a lejátszandó hangot 0-tól (mély) 127-ig (magas).





4 Sebesség

Milyen gyorsan és erősen szeretnéd „lenyomni a billentyűt"? A 0 olyan lágy, hogy valószínűleg nem fogsz hallani semmit, a 100 viszont alapértelmezésnek megfelel.

Üzenet módosítása

Most, hogy tudod, mi van egy Midi-üzenetben, nekiállhatsz kísérletezni: megváltoztathatod a lejátszandó hangot, hogy mennyi ideig kell kitartani azt, további hangokat adhatsz meg, sőt még a hangszert is lecserélheted.

1 A hang megváltoztatása

Próbálj ki egy számot 0 és 127 között a „hang be" és „hang ki" üzenetekben:

```
a.setMessage(144, 1, 20, 100);
```



2 A hang időtartamának megváltoztatása

Módosítsd úgy a „hang ki" eseményt (nem az *üzenetet*), hogy korábbi vagy későbbi ütemben következzen be:

```
b.setMessage(128, 1, 44, 100);
MidiEvent noteOff = new MidiEvent(b, 3);
```



3 A hangszer megváltoztatása

Adj egy új üzenetet a hanglejátszó üzenet ELÉ, amely az 1. csatorna hangszerét az alapértelmezett zongoráról valami másra cseréli. A „hangszercsere" üzenetet a „192" jelképezi, a használni kívánt hangszert pedig a harmadik argumentum (próbálj ki egy 0 és 127 közötti számot):

```
first.setMessage(192, 1, 102, 0);
```

„Hangszercsere" az 1. csatornán (1. zenész)

a 102-es hangszerre



2. változat: kísérletezés hangokkal parancssori argumentumok segítségével

Ez a változat is egyetlen hangot játszik le, de a hangszert és a hangjegyet itt parancssori argumentumokkal változtatod meg. Kísérletezz két, 0 és 127 közötti int érték átadásával. Az első int állítja be a hangszert, a második pedig a lejátszandó hangot.

```
import javax.sound.midi.*;

public class MiniMusicCmdLine {    // ez az első

    public static void main(String[] args) {
        MiniMusicCmdLine mini = new MiniMusicCmdLine();
        if (args.length < 2) {
System. out.println("Ne feledkezz meg a hangszert és a hangjegyet megadó argumentumokról!");
        } else {
            int instrument = Integer.parseInt(args[0]);
            int note = Integer.parseInt(args[1]);
            mini.play(instrument, note);
        }
    }  // a main lezárása

    public void play(int instrument, int note) {

        try {

            Sequencer player = MidiSystem.getSequencer();
            player.open();
            Sequence seq = new Sequence(Sequence.PPQ, 4);
            Track track = seq.createTrack();

            MidiEvent event = null;

            ShortMessage first = new ShortMessage();
            first.setMessage(192, 1, instrument, 0);
            MidiEvent changeInstrument = new MidiEvent(first, 1);
            track.add(changeInstrument);

            ShortMessage a = new ShortMessage();
            a.setMessage(144, 1, note, 100);
            MidiEvent noteOn = new MidiEvent(a, 1);
            track.add(noteOn);

            ShortMessage b = new ShortMessage();
            b.setMessage(128, 1, note, 100);
            MidiEvent noteOff = new MidiEvent(b, 16);
            track.add(noteOff);
            player.setSequence(seq);
            player.start();

        } catch (Exception ex) {ex.printStackTrace();}
    }  // a play lezárása
}  // az osztály lezárása
```

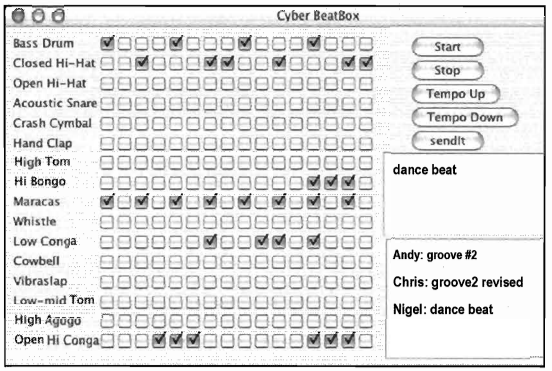
Futtasd a kódot két, 0 és 127 közötti int argumentummal. Kezdetnek próbálkozz ezekkel az értékekkel:

```
File Edit Window Help Attenuate
%java MiniMusicCmdLine 102 30
%java MiniMusicCmdLine 80 20
%java MiniMusicCmdLine 40 70
```

Mit főzünk ki a Kódkonyhában a következő fejezetekben?

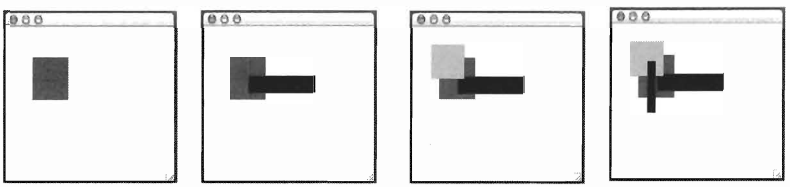
15. fejezet: a cél

Ha készen leszünk, lesz egy működőképes BeatBox-programod, amely egyben Drum Chat Client is. Ehhez azonban még tanulnod kell a grafikus felhasználói felületekről (GUI) – beleértve az eseménykezelést is –, a be- és kimenet kezeléséről (I/O), a hálózatkezelésről és a szálakról. A következő három fejezetben (12., 13. és 14. fejezet) mindent megtalálsz.



12. fejezet: MIDI-események

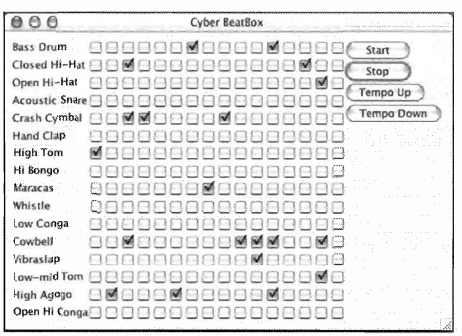
Ebben a Kódkonyhában egy apró „zenés videót" építünk (bár némi túlzás annak hívni...), amely véletlenszerű téglalapokat rajzol a MIDI-zene ütemére. Megtanulod, hogyan hozhatsz létre és „játszhatsz le" sok-sok MIDI-eseményt (nem csak néhányat, mint ebben a fejezetben).



első ütem **második ütem** **harmadik ütem** **negyedik ütem**

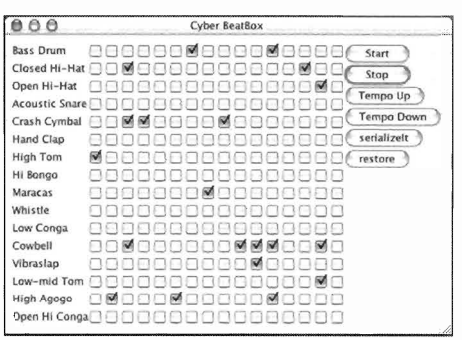
13. fejezet: Önálló BeatBox

Itt már ténylegesen felépítjük a BeatBox-programot, a grafikus felületet, meg mindent. A program azonban itt még korlátozott képességű lesz – amint megváltoztatod a mintát, elveszted az előzőt. Hiányzik még a mentés és a visszaállítás képessége, és a program nem kommunikál a hálózattal. (Arra azonban használhatod, hogy a saját dobmintáidat csiszolgasd.)



14. fejezet: Mentés és visszaállítás

Ha elkészítetted a tökéletes mintát, itt már fájlba mentheted és visszatöltheted, ha újra le szeretnéd játszani. Ezzel elő is készítjük a végső változatot (15. fejezet), amelyben a mintát fájlba írás helyett a hálózaton át elküldjük a csevegőkiszolgálónak.





Ez a fejezet feltárta előtted a kivételek csodálatos világát. A feladatod az, hogy eldöntsd, hogy a kivételekkel kapcsolatos alábbi állítások közül melyik igaz, és melyik hamis.

Igaz vagy hamis?

1. Egy try blokkot mindig egy catch *és* egy finally blokknak kell követnie.
2. Ha olyan tagfüggvényt írsz, amely a fordító által ellenőrzött kivételt válthat ki, a kockázatos kódot *kötelező* egy try/catch blokkba burkolnod.
3. A catch blokkok többalakúak is lehetnek.
4. Csak a fordító által ellenőrzött kivételeket lehet elkapni.
5. Ha try/catch blokkot határozol meg, nem kötelező annak megfelelő finally blokkot is megadnod.
6. Ha meghatározol egy try blokkot, egy megfelelő catch vagy finally blokkal, vagy mindkettővel párosíthatod.
7. Ha olyan tagfüggvényt írsz, amely bejelenti, hogy fordító által ellenőrzött kivételt válthat ki, a kivételt dobó kódot is egy try/catch blokkba kell burkolnod.
8. A program main() tagfüggvényének minden neki dobott kezeletlen kivételt kezelnie kell.
9. Egyetlen try blokkhoz több különböző catch blokk is tartozhat.
10. Egy tagfüggvény csak egyféle kivételt válthat ki.
11. A finally blokk mindenképpen lefut, függetlenül attól, hogy fellépett-e kivétel.
12. Egy finally blokk try blokk nélkül is létezhet.
13. Egy try blokk állhat önmagában, catch vagy finally blokk nélkül is.
14. A kivételek kezelését néha „lebukásnak" hívják.
15. A catch blokkok sorrendje soha nem számít.
16. Egy try és egy finally blokkal rendelkező tagfüggvény azt is megteheti, hogy bejelenti, hogy lebukik a kivétel elől.
17. A futásidejű kivételeket mindenképpen *kezelni kell*, vagy *le kell bukni előlük.*



Gyakorlat



Kódmágnesek

Egy működőképes Java-program kódja összekeveredett a hűtőmágneseken. Össze tudod rakni úgy a részleteket, hogy előálljon belőlük a lap alján látható kimenetet eredményező Java-program? Néhány kapcsos zárójel leesett a padlóra, és túl kicsi volt ahhoz, hogy fel tudjuk venni, úgyhogy nyugodtan egészítsd ki a kódot annyi kapcsos zárójellel, amennyit csak szükségesnek érzel!

```
System.out.print("r");
try {
doRisky(test);
System.out.print("t");
System.out.println("s");
} finally {
System.out.print("o");
class MyEx extends Exception { }
public class ExTestDrive {
System.out.print("w");
if ("yes".equals(t)) {
System.out.print("a");
throw new MyEx();
} catch (MyEx e) {
static void doRisky(String t) throws MyEx {
  System.out.print("h");
public static void main(String [] args) {
  String test = args[0];
```

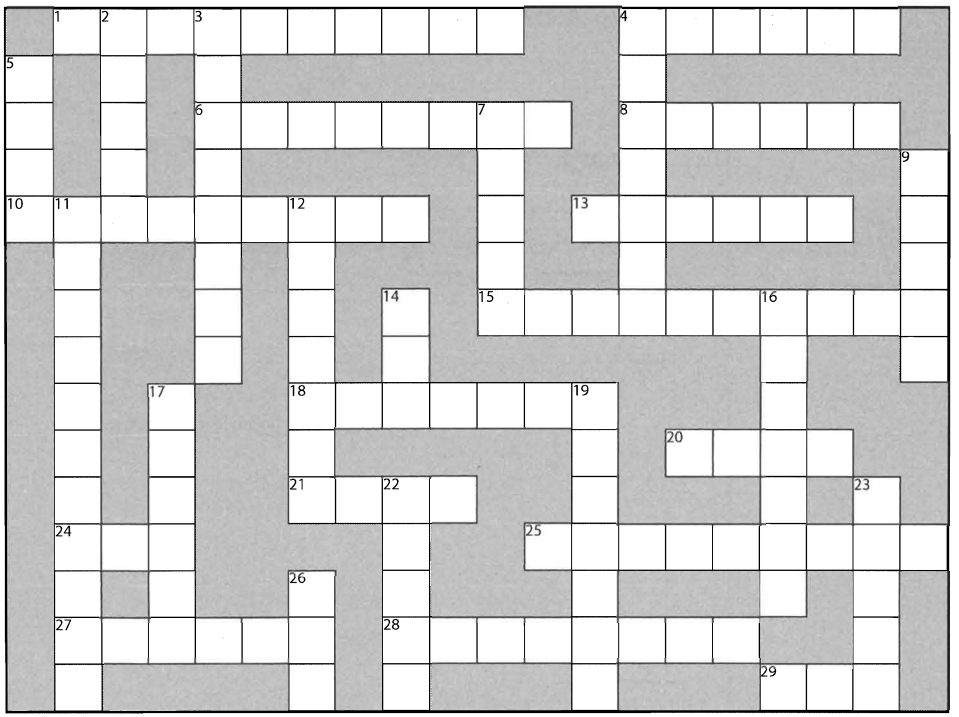
```
File Edit Window Help ThrowUp
% java ExTestDrive yes
thaws

% java ExTestDrive no
throws
```



Java-keresztrejtvény 7.0

Tudod, mit kell tenned!



Vízszintes

1. Értékadás (angolul, a másik szóval)
4. Lepattant a tetejéről (angolul, múlt időben)
6. Mindent tud, amit az ősе, és még többet is (angolul)
8. Elindít (másik szóval, angolul)
10. Családfa (másik szóval, angolul)
13. Nem bukik le (angolul)
15. Problémás objektumok (többesszámban, angolul)
18. A Javában 49 ilyen van (angolul)
20. Osztályhierarchia (másik szóval, angolul)
21. Ezt csinálod, ha túl forró megfogni (angolul)
24. Gyakran használt alaptípus
25. Kódrecept (angolul)
27. Ezt csinálja egy rendetlen tagfüggvény
28. Nem Picasso stílusa (angolul)
29. Eseménysort indít el

Függőleges

2. Ez határozza meg, hogy éppen mi használható (angolul)
3. Egyfajta sablonból jön létre (angolul)
4. A gyerekek nem láthatják
5. Nagyrészt statikus API-osztály
7. Nem viselkedés (angolul)
9. A sablon (angolul)
11. Legördít még egyet a futószalagról (angolul)
12. A javac látja az ilyen kivételeket (angolul)
14. Kockázatos kísérlet
16. Automatikusan hozzájut (angolul)
17. Módosítófüggvény (angolul)
19. Bevezet (a másik szóval, angolul)
22. Elkap, angolul
23. Rossz hírt közöl
26. Egyfajta

További tippek:

Vízszintes

6. Java-gyermek
8. Tagfüggvényt indít
13. Beágyazott helyzet
20. Van ilyen gyűjtemény is
21. Ha angolul van, hát pofhat is
27. Újjára enged egy problémát
28. Nem elvont

Függőleges

2. szteto, horo…
3. for example más-képp: for_____
5. Számok…
9. Csak public vagy default lehet
16. _____ a családi vagyont
17. Nem getter („kiolvasó”)



Gyakorlat – Megoldás

Igaz vagy hamis?

1. Hamis: vagy csak az egyik, vagy mindkettő.
2. Hamis: be is jelentheted a kivételt (amivel lebuksz előle).
3. Igaz.
4. Hamis: a futásidejű kivételeket is el lehet kapni.
5. Igaz.
6. Igaz: mindkettő elfogadható.
7. Hamis: a bejelentés elég.
8. Hamis, de ha nem teszi, a JVM lefagyhat.
9. Igaz.
10. Hamis.
11. Igaz: gyakran arra használják, hogy rendet rakjon a félig befejezett feladatok után.
12. Hamis.
13. Hamis.
14. Hamis: a „lebukás” a „bejelentés” szinonimája.
15. Hamis: a legtágabb kivételt az utolsó catch blokknak kell elkapnia.
16. Hamis: ha nincs catch blokkod, kötelező bejelentened (lebuknod).
17. Hamis.

Kódmágnesek

```
class MyEx extends Exception { }

public class ExTestDrive {

  public static void main(String [] args) {
    String test = args[0];
    try {

       System.out.print("t");

       doRisky(test);

       System.out.print("o");

    } catch ( MyEx e) {

        System.out.print("a");

    } finally {

        System.out.print("w");
    }
    System.out.println("s");
  }

  static void doRisky(String t) throws MyEx {
    System.out.print("h");

    if ("yes".equals(t)) {

      throw new MyEx();
    }

    System.out.print("r");

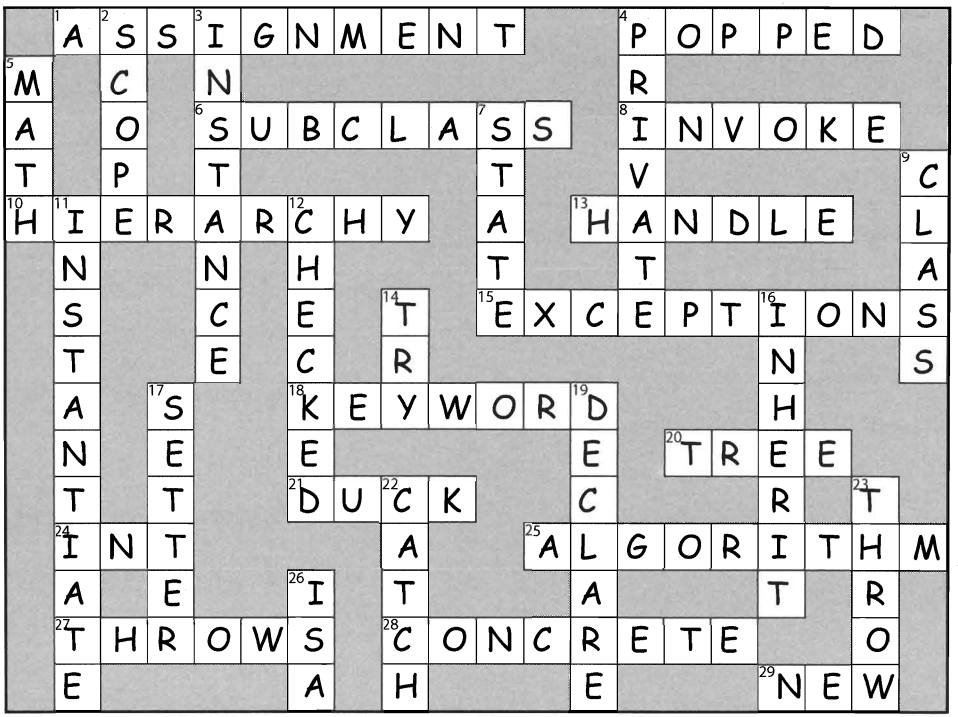
  }
}
```

```
File Edit Window Help Chill
% java ExTestDrive yes
thaws

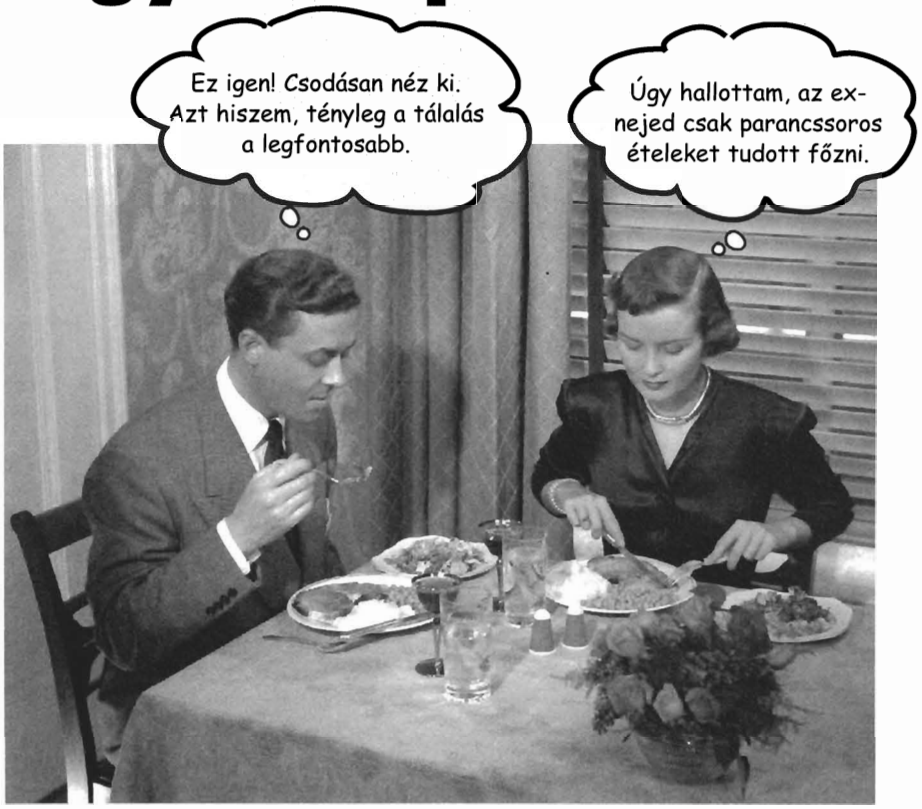
% java ExTestDrive no
throws
```



Java-keresztrejtvény – Megoldás



Egy nagyon képszerű történet



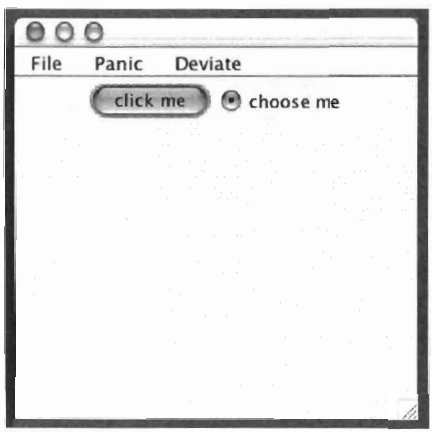
Nézz szembe a ténnyel: szükséged lesz grafikus felületek építésére. Ha olyan alkalmazásokat készítesz, amelyeket mások fognak használni, mindenképpen *kelleni fognak* grafikus felületek. Ha magadnak készítesz programokat, akkor meg *te* akarsz majd grafikus felületet. Még ha meg is vagy győződve róla, hogy életed hátralevő részét kiszolgálóoldali kódok írásával fogod tölteni, ahol az ügyféloldali felhasználói felület egy weboldal, előbb-utóbb akkor is eszközöket kell majd készítened, és szükséged lesz egy grafikus felületre. Egy parancssori alkalmazás persze nagyon „retró" – de nem éppen a jó értelemben: gyenge, rugalmatlan, barátságtalan. Ezért két fejezetet is grafikus felhasználói felületek készítésének szentelünk, és közben megtanuljuk használni a Java néhány kulcsfontosságú szolgáltatását, beleértve az **eseménykezelést** és a **belső osztályokat**. Ebben a fejezetben egy gombot helyezünk el a képernyőn, és végrehajtatunk vele valamit, amikor a felhasználó rákattint; ezenkívül rajzolunk a képernyőre, megjelenítünk egy jpeg képet, és még animációt is készítünk.

Minden egy ablakkal kezdődik

A JFrame-ek (Java-keretek) azok az objektumok, amelyek egy-egy ablakot jelképeznek a képernyőn. Bennük kell elhelyezned minden olyan felületi elemet, mint a gombok, jelölőnégyzetek, szövegmezők, és így tovább. Lehet rajtuk egyszerű menüsor és azok a kis – az adott rendszernek megfelelő – ablakkezelő ikonok, amelyekkel lekicsinyítheted, felnagyíthatod vagy bezárhatod az ablakot. A JFrame más-más külsőt ölt attól függően, hogy milyen rendszeren dolgozol. Mac OS X-en így néz ki:



„Ha meglátok még egy parancssoros alkalmazást, ki vagy rúgva."



JFrame menüsorral és két „kütyüvel" (egy gombbal és egy választógombbal)

Kütyük az ablakban

Ha már van egy JFrame-ed, vezérlőket („kütyüket", widget) helyezhetsz el benne. Egy halom Swing-összetevő létezik, amit egy ablakhoz adhatsz; ezeket a javax.swing csomagban találod. A leggyakrabban használtak a JButton (gomb), a JRadioButton (választógomb), a JCheckBox (jelölőnégyzet), a JLabel (szövegcímke), a JList (listamező), a JScrollPane (gördíthető mező), a JSlider (csúszka), a JTextArea (többsoros szövegmező), a JTextField (egysoros szövegmező) és a JTable (táblázat). A legtöbbet nagyon könnyű használni, bár van olyan is (például a JTable), amit egy kicsit bonyolultabb.

Grafikus felületet készíteni könnyű:

1. Létrehozol egy keretet (JFrame-et):

```
JFrame frame = new JFrame();
```

2. Létrehozol egy vezérlőt (gombot, szövegmezőt stb.):

```
JButton button = new
JButton("Kattints rám!");
```

3. A vezérlőt a kerethez adod:

```
frame.getContentPane().add(button);
```

A kerethez nem közvetlenül adsz dolgokat. A keretre gondolj úgy, mint az ablak körüli szegélyre – az elemeket az ablaktáblához (pane) kell hozzáadnod.

4. Megjeleníted a keretet (megadod a méretét, és láthatóvá teszed):

```
frame.setSize(300,300);
frame.setVisible(true);
```

Az első grafikus felületed: gomb a keretben

```
import javax.swing.*;

public class SimpleGui1 {
    public static void main (String[] args) {

        JFrame frame = new JFrame();
        JButton button = new JButton("click me");


        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);


        frame.getContentPane().add(button);


        frame.setSize(300,300);


        frame.setVisible(true);
    }
}
```

ne felejtsd el betölteni ezt a swing-csomagot

létrehozunk egy keretet és egy gombot (a gombon megjeleníteni kívánt szöveget a gomb konstruktorának adhatod át)

ez a sor kilépteti a programot, amint bezárod az ablakot (ha ezt kihagyod, az ablak örökké ott marad a képernyőn)

a gombot hozzáadjuk a keret tartalomtáblájához

megadjuk a keret méretét, képpontban

végül láthatóvá tesszük az ablakot (ha megfeledkezel erről a lépésről, a kód futtatásakor semmit nem fogsz látni)

Lássuk, mi történik, ha futtatod a fenti kódot:

```
%java SimpleGui1
```



Húha! Ez aztán Nagy Gomb!

A gomb a keretben rendelkezésre álló teljes területet kitölti. Később megtanulod, hogyan szabályozhatod a gomb helyét (és nagyságát) a keretben.

De hát semmi nem történik, amikor a gombra kattintok...

Ez nem teljesen igaz. Amikor megnyomod a gombot, a kinézete „benyomottra" változik (a külső a rendszertől függ, de *valami* mindig jelzi, hogy a gomb be van nyomva).
A valódi kérdés ez: „Hogyan vehetem rá a gombot, hogy egy bizonyos dolgot csináljon, amikor a felhasználó rákattint?".

Két dologra van szükséged:

1. Kell egy ***tagfüggvény***, ami akkor hívódik meg, amikor a felhasználó a gombra kattint (ez fogja végrehajtani azt a műveletet, amit a gombra kattintás eredményeként szeretnél).
2. Valahonnan ***tudnod*** kell, hogy mikor kell elindítani az említett tagfüggvényt. Más szavakkal, valahonnan tudnod kell, mikor kattint a felhasználó.



Tudnod kell róla, ha a felhasználó kattint. Egy olyan eseményre van szükség, ami azt jelképezi, hogy a felhasználó valamilyen <u>műveletet</u> hajtott végre egy gombon.

Nincsenek
hülye kérdések

K: A gomb úgy fog kinézni, mint a Windows gombjai, ha Windowson futtatom a programot?

V: Ha akarod, igen. Az alapkönyvtárban több stílusosztály közül választhatsz, amelyek a felület megjelenítését és működését („look-and-feel") szabályozzák. A legtöbb esetben legalább két különböző stílus áll rendelkezésre: a szabványos Java-stílus, amelyet ***Metal*** („Fém") néven is ismernek, illetve a rendszer alapértelmezett stílusa. Az ebben a könyvben szereplő Mac OS X-képernyőfelvételeken vagy az OS X ***Aqua*** stílusa, vagy a ***Metal*** stílus látható.

K: Megoldható, hogy egy program mindig Aqua stílusban jelenjen meg, akkor is, ha Windowson fut?

V: Sajnos nem. Nem minden stílus érhető el minden rendszeren. Ha biztonságra törekszel, vagy állítsd be kifejezetten a Metal stílust, hogy pontosan tudd, mit kapsz, függetlenül attól, hogy hol fut a program, vagy ne adj meg stílust, csak fogadd el az alapértelmezést.

K: Azt hallottam, hogy a Swing csigalassúságú, ezért senki sem használja.

V: Ez régebben így is volt, de ma már nem feltétlenül igaz. Szerényebb képességű számítógépeken fájdalmas lehet a Swing használata, az újabb asztali rendszereken azonban – főleg a Java 1.3-as és későbbi változataiban – valószínűleg észre sem veszed a különbséget egy Swing-felület és a rendszer saját grafikus felületei között. A Swinget ma már rengeteg különböző fajta alkalmazásban használják.

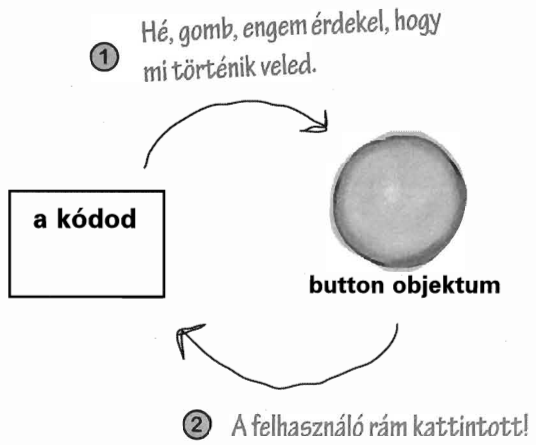
Értesülés a felhasználói eseményekről

Tegyük fel, hogy azt szeretnéd, ha a gomb szövege *Kattints rám*-ról (click me) *Rám kattintottak*-ra (I've been clicked) változna, amikor a felhasználó megnyomja a gombot. Először írj egy tagfüggvényt, ami megváltoztatja a gomb szövegét (egy gyors keresés az API-ban felfedi a tagfüggvényt):

```
public void changeIt() {
    button.setText("Rám kattintottak!");
}
```

És *innen* hogyan tovább? Honnan tudhatod, hogy *mikor* kell lefutnia ennek a tagfüggvénynek? ***Honnan fogod tudni, ha a gombra kattintottak?***
A Javában a felhasználói eseményekről való értesülést és azok kezelését *eseménykezelésnek* (event-handling) hívják. A Javában számos különféle eseménytípus létezik, bár a legtöbb a grafikus felületen végrehajtott felhasználói műveletekhez kapcsolódik. Ha a felhasználó egy gombra kattint, az egy esemény, mégpedig egy olyan esemény, ami azt mondja, hogy „a felhasználó ennek a gombnak a műveletét szeretné végrehajtatni". Ha mondjuk a Slow Tempo (Lassú tempó) gombról van szó, a felhasználó azt akarja, hogy a tempó lelassuljon; ha egy Send (Küldés) gombról egy csevegőprogramban, akkor pedig nyilván üzenetet akar küldeni. A legegyszerűbb esemény tehát az, amikor a felhasználó egy gombra kattint, jelezve, hogy végre akarja hajtatni a hozzá kapcsolódó műveletet.
A gombok esetében számodra általában nem érdekesek az olyan köztes események, mint a gomb lenyomása és felengedése, mivel csak annyit akarsz mondani a gombnak, hogy „Nem érdekel, hogyan játszadozik a felhasználó a gombbal, mennyi ideig tartja felette az egeret, hányszor gondolja meg magát, és húzza arrébb, mielőtt mégis rákattintana – csak értesíts, amikor tényleg megtette! Vagyis ne hívj, csak ha a felhasználó olyan módon kattint, ami azt jelzi, hogy a feliratának megfelelően akarja működtetni azt a nyomorult gombot!".

Először is, a gombnak tudnia kell, hogy gondolsz rá.



Másodszor, a gombnak valahogy vissza kell hívnia téged, amikor rákattintanak, vagyis bekövetkezik egy „gombra kattintás" esemény.

Agytorna

1) Hogyan mondhatod meg egy gomb objektumnak, hogy értesülni szeretnél az eseményeiről? Hogyan közölheted vele, hogy figyeled?

2) Hogyan hívhat vissza téged a gomb? Tételezzük fel, hogy nincs mód rá, hogy közöld a gombbal az egyedi tagfüggvényed nevét (changeIt()). Milyen más módon biztosíthatod a gombot, hogy van egy tagfüggvényed, amelyet meghívhat, amikor az esemény bekövetkezik? (Tipp: gondolj a házikedvencekre!)

Ha értesülni szeretnél egy gomb eseményeiről, valósíts meg egy felületet, ami azt mondja, hogy „Figyelem az eseményeidet.".

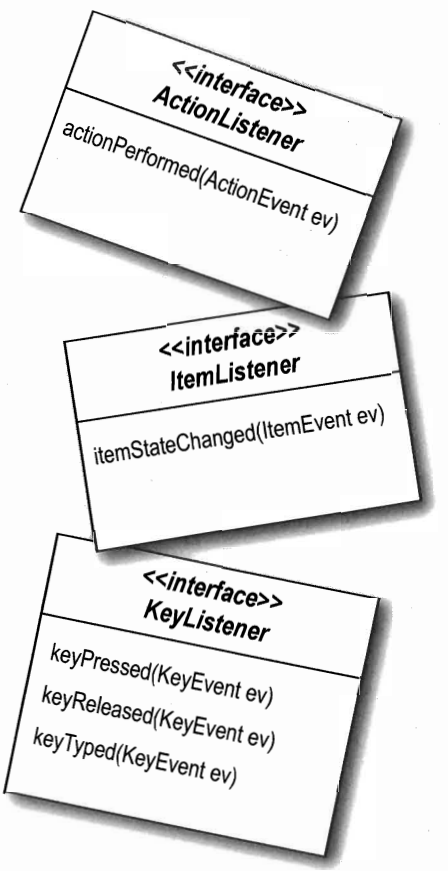
A figyelőfelület (listener interface) a híd a figyelő (listener – te) és az esemény forrása (event source – a gomb) között.

A Swing grafikus felületi összetevői eseményforrások. A Java szóhasználatában az eseményforrás olyan objektum, amely a felhasználói műveleteket (egérkattintás, billentyűleütés, ablak bezárása) eseményekké tudja alakítani. Mint szinte minden mást, a Javában az eseményeket is objektumok, méghozzá valamilyen eseményosztályba tartozó objektumok ábrázolják. Ha végigböngészed a java.awt.event csomagot az API-ban, egy csomó eseményosztályt láthatsz (könnyű észrevenni őket, mert mindegyiknek a nevében megtalálható az ***Event*** szó): MouseEvent (egéresemény), KeyEvent (billentyűesemény), WindowEvent (ablakesemény), ActionEvent (műveletesemény), és így tovább.

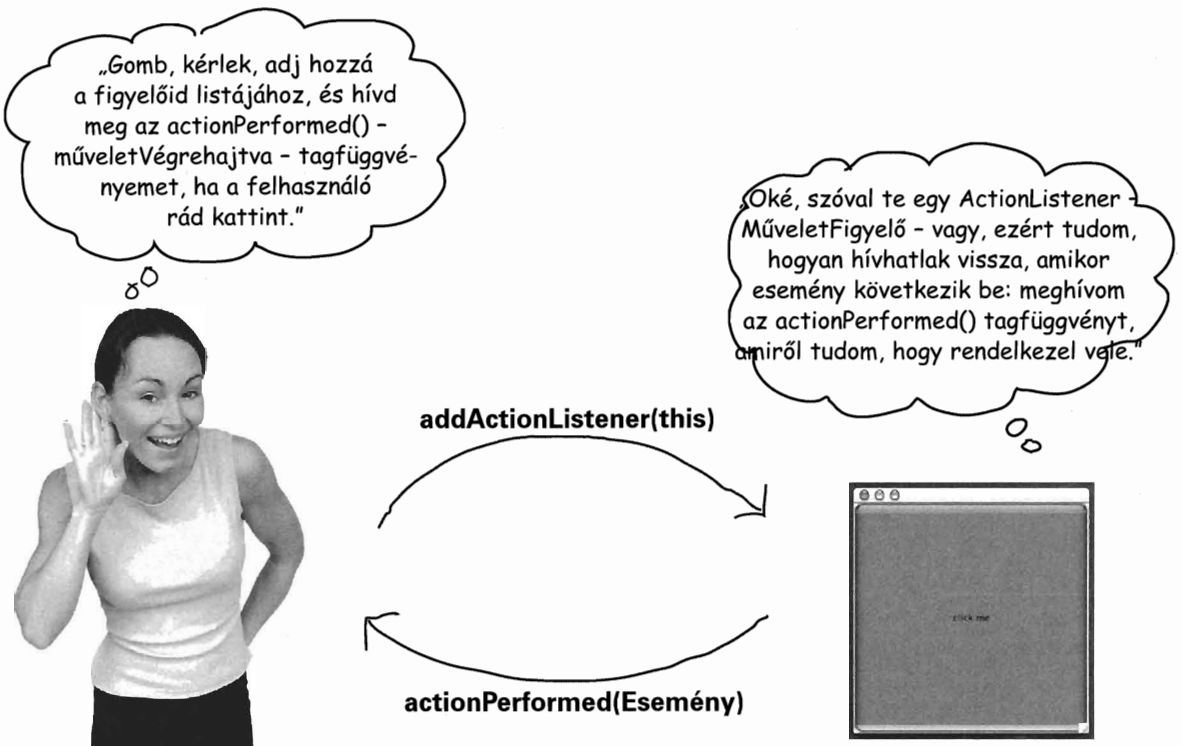
Az ***eseményforrás*** (például egy gomb) egy ***eseményobjektumot*** (event object) hoz létre, amikor a felhasználó valami olyasmit csinál, ami fontos a program számára (például a gombra *kattint*). Az általad írt kódok többsége (és az összes kód ebben a könyvben) inkább *fogadja*, mintsem *létrehozza* az eseményeket. Más szavakkal, többnyire esemény*figyelőként*, és nem esemény*forrásként* fogsz szerepelni. Minden eseménytípushoz tartozik egy neki megfelelő figyelőfelület. Ha egéreseményeket szeretnél, valósítsd meg a MouseListener felületet; ha ablakeseményeket, akkor a WindowListener-t, és így tovább. És ne feledkezz meg a felületek szabályairól: amelyik felületről *bejelented* (class Dog implements Pet) hogy megvalósítod, azt meg is kell valósítanod, ami azt jelenti, hogy meg kell írnod a felület valamennyi tagfüggvényének megvalósítását.

Egyes felületek egynél több tagfüggvénnyel rendelkeznek, mert magának az eseménynek többféle változata is bekövetkezhet. Például ha a MouseListener felületet valósítod meg, az egérgomb lenyomását (mousePressed) és felengedését (mouseReleased), az egér mozgatását (mouseMoved) stb. jelképező eseményekkel találkozhatsz, és ezekhez az egéreseményekhez külön-külön tagfüggvény tartozik a felületben, annak ellenére, hogy mind egy MouseEvent eseményt vár. Ha megvalósítod a MouseListener felületet, a mousePressed() tagfüggvény akkor hívódik meg, amikor – nyilván kitaláltad – a felhasználó lenyomja az egér gombját, a mouseReleased() pedig akkor, amikor felengedi. Az egéreseményeket tehát egyetlen esemény*objektum*, a MouseEvent jelképezi, de több különböző esemény*függvény*, amelyek a különböző típusú egéreseményeket jelölik.

Amikor megvalósítod a figyelőfelületet, módot adsz a gombnak arra, hogy visszahívjon. A felület (interface) az a hely, ahol a visszahívó (callback) tagfüggvényt bevezeted.



Hogyan társalog a figyelő és a forrás?



A Figyelő

Ha az osztályod értesülni szeretne egy gomb műveleti eseményeiről (ActionEvent), akkor meg kell valósítanod az ActionListener felületet. A gombnak tudnia kell róla, hogy figyeled, ezért be kell jelentkezned a gombnál a gomb addActionListener(this) tagfüggvényének meghívásával, átadva annak egy ActionListener hivatkozást (ebben az esetben *te* vagy a műveletfigyelő, ezért a *this*-t adod át). A gombnak valahogy vissza kell tudnia hívni téged, amikor az esemény bekövetkezik, ezért meghívja a figyelőfelület tagfüggvényét. Műveletfigyelőként *kötelező* megvalósítanod a felület egyetlen, actionPerformed() nevű tagfüggvényét. A fordító betartatja ezt a követelményt.

Az Eseményforrás

A gombok műveleti események forrásai, ezért tudniuk kell, mely objektumok figyelik őket. A gombnak van egy addActionListener() tagfüggvénye, amely módot ad az érdeklődő objektumoknak (a figyelőknek) arra, hogy *közöljék* a gombbal, hogy figyelik. Amikor a gomb addActionListener() tagfüggvénye lefut (mert egy figyelő meghívta), a gomb átveszi a paraméterét (a figyelőobjektumra mutató hivatkozást), és elraktározza egy listában. Ha a felhasználó a gombra kattint, a gomb „elindítja" („elsüti", fire) az eseményt, mégpedig úgy, hogy meghívja az actionPerformed() tagfüggvényt a listájában szereplő összes figyelőre.

Hogyan értesülhetsz egy gomb műveleti eseményéről?

1. Megvalósítod az ActionListener felületet
2. Bejelentkezel a gombnál (közlöd vele, hogy figyelni akarod az eseményeit)
3. Meghatározod az eseménykezelő tagfüggvényt (megvalósítod az ActionListener felület actionPerformed() tagfüggvényét)

```
import javax.swing.*;
import java.awt.event.*;

public class SimpleGui1B implements ActionListener {
    JButton button;

    public static void main (String[] args) {
       SimpleGui1B gui = new SimpleGui1B();
       gui.go();
     }

    public void go() {
       JFrame frame = new JFrame();
       button = new JButton("click me");

       button.addActionListener(this);

       frame.getContentPane().add(button);
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       frame.setSize(300,300);
       frame.setVisible(true);
    }

    public void actionPerformed(ActionEvent event) {
       button.setText("I've been clicked!");
    }
}
```

Új betöltési utasítás az ActionListener-t és az ActionEvent-et tároló csomaghoz.

1

A felület megvalósítása. Ez azt mondja, hogy „a SimpleGui1b minden példánya EGYFAJTA ActionListener". (A gomb csak az ActionListener megvalósítóinak ad át eseményeket.)

2

Az érdeklődés jelzése a gombnak. Ez azt mondja a gombnak, hogy „adj hozzá a figyelőid listájához". Az átadott argumentumnak KÖTELEZŐ olyan osztályból származó objektumnak lennie, amely megvalósítja az ActionListener felületet!!!

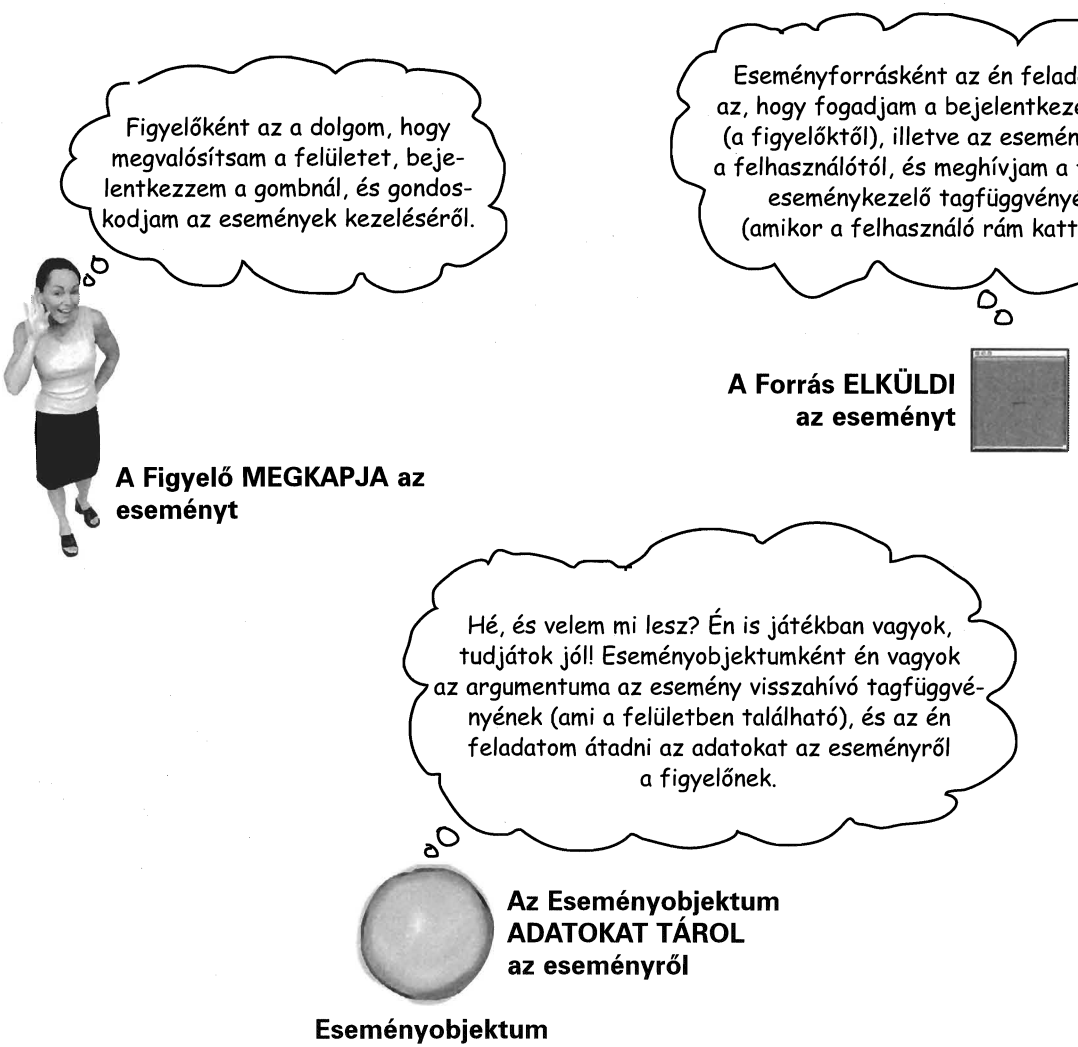
3

Az ActionListener felület actionPerformed() tagfüggvényének megvalósítása. Ez a tényleges eseménykezelő tagfüggvény.

A gomb ezt a tagfüggvényt hívja meg, hogy értesítsen egy esemény bekövetkeztéről. Argumentumként egy ActionEvent objektumot kapsz tőle, de erre nincs szükséged. Elég tudnod, hogy az esemény bekövetkezett.

Figyelők, források és események

Tündöklő Java-pályafutásod nagy részében nem *te* leszel az események *forrása*. (Nem számít, mennyire szeretnéd, hogy minden körülötted forogjon.)
Szokj hozzá a gondolathoz. A ***te feladatod az, hogy figyelmes hallgató legyél***. (Ami – ha őszintén igyekszel – *javíthat* a megítéléseden.)



K: Én miért nem lehetek eseményforrás?

V: LEHETSZ. Mi csak azt mondtuk, hogy te a *legtöbbször* az esemény fogadója, és nem a kezdeményezője leszel (legalábbis csodálatos Java-pályafutásod *elején*). A számodra érdekes események legtöbbjét a Java API-ban található osztályok indítják el; neked csak annyit kell tenned, hogy figyeled őket. Ugyanakkor tervezhetsz olyan programot, amelyben mondjuk arra van szükséged, hogy egy StockMarketEvent (ÉrtékTőzsdeEsemény) nevű egyedi esemény következzen be, amikor a tőzsdefigyelő alkalmazás valami olyasmit vesz észre, amit fontosnak ítél. Ebben az esetben a StockWatcher (TőzsdeFigyelő) objektumot kell eseményforrássá tenned, és ugyanazokat a dolgokat kell végrehajtanod, mint amit egy gomb (vagy bármilyen más forrás) csinál – létre kell hoznod egy figyelőfelületet az egyedi eseményedhez, biztosítanod kell egy bejegyzőfüggvényt (addStockListener()), és amikor valaki meghívja azt, hozzá kell adnod a hívót (egy figyelőt) a figyelők listájához. Ezután, ha bekövetkezik valamilyen tőzsdei esemény, létre kell hoznod egy StockEvent-objektumpéldányt (ezt az osztályt is neked kell megírnod), és át kell adnod a listádban szereplő figyelőknek azok stockChanged(StockEvent) – tőzsdeMegváltozott(TőzsdeEsemény) – tagfüggvényének meghívásával. Ezenkívül azt sem szabad elfelejtened, hogy minden *eseménytípushoz* rendelkezned kell *megfelelő figyelőfelülettel* (tehát kell egy stockChanged() tagfüggvénnyel rendelkező StockListener felület).

K: Nem értem az esemény visszahívó tagfüggvényeinek átadott eseményobjektum jelentőségét. Ha valaki meghívja a mousePressed tagfüggvényemet, milyen további információkra lehet szükségem?

V: A legtöbb esetben nincs szükséged az eseményobjektumra. Az nem több egy apró adathordozónál, amely további információkat közöl az eseményről. Néha azonban konkrét részleteket kell megkérdezned az eseménytől. Például ha a mousePressed() tagfüggvényedet hívják, tudod, hogy lenyomták az egérgombot – de mi van, ha pontosan tudni szeretnéd, hogy ez hol történt? Más szavakkal, hogyan derítheted ki az egérgomb lenyomásának X és Y koordinátáit?

Az is előfordulhat, hogy *ugyanazt* a figyelőt *több* objektum listájába is be szeretnéd jegyeztetni. Egy képernyőn megjelenített számológépnek például 10 számgombja van, és mivel mind ugyanazt csinálja, nem érdemes külön figyelőt létrehozni minden egyes gombhoz. Ehelyett elég, ha egyetlen figyelőt jegyeztetsz be a 10 gomb mindegyikével, és amikor eseményre kerül sor (vagyis meghívják a visszahívó tagfüggvényedet), meghívsz egy tagfüggvényt az eseményobjektumra, hogy kiderítsd, *melyik gomb* volt az adott esemény tényleges forrása.

Nem voltak

hülye kérdések

Hogyan TUDHATOM hogy egy objektum eseményforrás-e?

Nézd meg az API-ban.

OK. Mit keressek?

Egy olyan tagfüggvényt, amelynek a neve „add"-dal kezdődik, „Listener"-rel végződik, és argumentumként egy figyelőfelületet vár. Ha ezt látod: addKeyListener(KeyListener k) ...akkor tudhatod, hogy azok az osztályok, amelyekben megtalálható ez a tagfüggvény, billentyűesemények (KeyEvent) forrásai. A név mindent elárul.

Hegyezd ki a ceruzád!

Az alábbi vezérlők (felhasználói felületi objektumok) mindegyike egy vagy több esemény forrása. Párosítsd össze a vezérlőket azokkal az eseményekkel, amelyeket kiválthatnak. Egyes vezérlők egynél több esemény forrásai is lehetnek, és egyes eseményeket egynél több vezérlő is kiválthat.

Vezérlők	Esemény-tagfüggvények
jelölőnégyzet	windowClosing()
szövegmező	actionPerformed()
gördíthető lista	itemStateChanged()
gomb	mousePressed()
párbeszédablak	keyTyped()
választógomb	mouseExited()
menüelem	focusGained()

A rajzolásra visszatérve...

Most, hogy már tudsz ezt-azt az események működéséről (később még többet is meg fogsz tudni), térjünk vissza ahhoz, hogy miként rajzolhatsz dolgokat a képernyőre. Pár percet eljátszadozunk a rajzolással, mielőtt a figyelmünket ismét az eseménykezelésre fordítanánk.

A grafikus felületre rajzolás három módja:

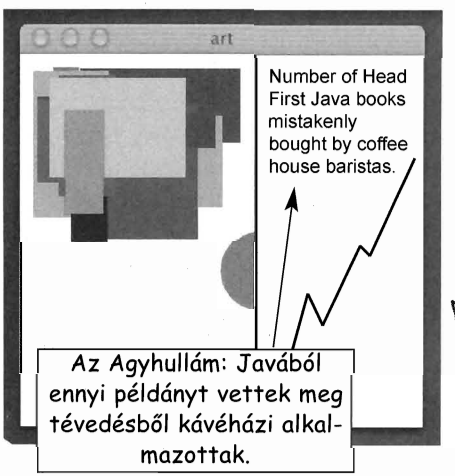


1 Vezérlők elhelyezése egy keretben

Gombok, menük, választógombok stb. hozzáadása:

```
frame.getContentPane().add(myButton);
```

A javax.swing csomag egy tucatnál is több vezérlőtípust tartalmaz.



2 Kétdimenziós grafika rajzolása egy vezérlőre

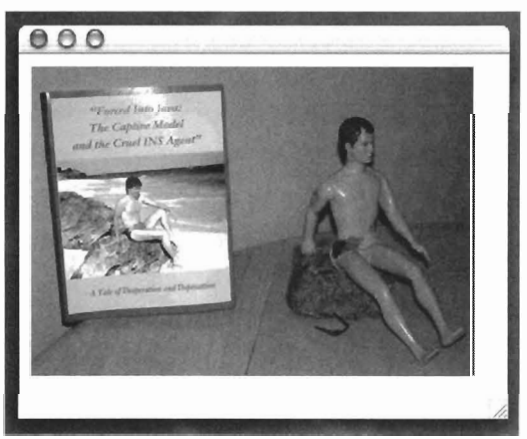
Alakzatokat egy grafikai objektum segítségével rajzolhatsz:

```
graphics.fillOval(70,70,100,100);
```

Nem csak négyszögeket és köröket lehet rajzolni: a Java2D API tele van hasznos és kifinomult grafikai függvényekkel.

műalkotások, játékok, szimulációk stb.

grafikonok, üzleti diagramok stb.



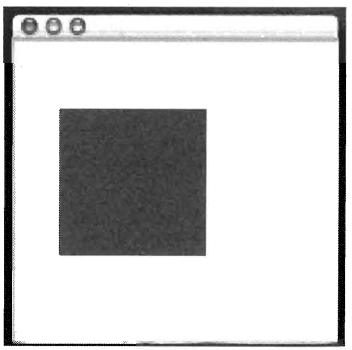
3 JPEG-kép elhelyezése egy vezérlőn

A vezérlőkön saját képeket is elhelyezhetsz:

```
graphics.drawImage(myPic,10,10,this);
```

Készíts saját rajztáblát!

Ha saját grafikát szeretnél megjeleníteni a képernyőn, célszerű létrehoznod egy saját vezérlőt, amelyre festhetsz vagy rajzolhatsz. Ezt a vezérlőt aztán elhelyezheted egy keretben – ugyanúgy, mint egy gombot vagy bármely más vezérlőt –, és amikor megjelenik, ott lesznek rajta a képeid. A képeket mozgásba is hozhatod (animálhatod), vagy éppen gombnyomásra megváltoztathatod a színeket.
Pofonegyszerű!



Hozd létre a JPanel egy alosztályát, és írd felül a paintComponent() tagfüggvényt.

Minden grafikai kódodat a paintComponent() tagfüggvényben kell elhelyezned, amelyre úgy gondolhatsz, mint aminek a meghívásával a rendszer azt mondja, hogy „Hé, vezérlő, ideje kirajzolnod magad!". Ha mondjuk egy kört szeretnél rajzolni, a paintComponent() tagfüggvényben a kör megrajzolásához szükséges kódot kell elhelyezned. Amikor a rajztábládat tartalmazó keret megjelenik, meghívódik a paintComponent() tagfüggvény, és kirajzolja a kört. Ha a felhasználó ikonná zsugorítja az ablakot, a JVM tudja, hogy a keretnek „javításra" lesz szüksége, ha ismét felnagyítják, ezért újra meghívja a paintComponent()-et. Minden alkalommal, amikor a JVM úgy gondolja, hogy a képernyőnek frissítésre van szüksége, meghívja a paintComponent() tagfüggvényedet.
Egy dolgot azonban az eszedbe kell vésned: ***ezt a tagfüggvényt soha nem te magad hívod meg***! A tagfüggvény argumentuma (egy Graphics objektum) a tényleges rajzvászon, amelyet a Java a *valódi* megjelenítőre fektet. Ezt nem érheted el önállóan – a feladatot át kell adnod a rendszernek. Mindazonáltal később látni fogod, hogy *megkérheted* a rendszert, hogy frissítse a képernyőt – repaint() –, ami végeredményben a paintComponent() meghívásához vezet.

```
import java.awt.*;
import javax.swing.*;

class MyDrawPanel extends JPanel {

    public void paintComponent(Graphics g) {

        g.setColor(Color.orange);

        g.fillRect(20,50,100,100);

    }
}
```

mindkettőre szükséged lesz

Létrehozod a JPanel egy alosztályát, egy vezérlőt, amelyet ugyanúgy egy kerethez adhatsz, mint bármi mást – csak éppen ez a saját, testreszabott vezérlőd lesz.

Ez a Legfontosabb Grafikai Tagfüggvény. SOHA nem te magad hívod meg, hanem a rendszer, ami ezzel azt mondja, hogy „Itt egy friss, tiszta, Graphics típusú rajzfelület, amelyre most már festhetsz".

Képzeld azt, hogy a „g" egy festőgép. Megmondod neki, hogy milyen színnel fessen, majd azt, hogy milyen alakzatot (a helyét megszabó koordinátákkal, illetve a méretével).

Ilyen klassz dolgokat csinálhatsz a paintComponent() segítségével

Nézzünk meg még pár dolgot, amit a paintComponent() tagfüggvénnyel csinálhatsz! A legszórakoztatóbb persze az, amikor saját magad kísérletezel. Játszadozz el a számokkal, és nézd meg az API-ban, hogy mit tud a Graphics osztály. (Később azt is látni fogod, hogy a Graphics osztály lehetőségein túl még *sokkal többet* is csinálhatsz.)

JPEG-kép megjelenítése

```
public void paintComponent(Graphics g) {

    Image image = new ImageIcon("catzilla.jpg").getImage();

    g.drawImage(image,3,4,this);

}
```

ide jön a fájlnév

Ezek az x,y koordináták a kép bal felső sarkának koordinátáit adják meg. Az itt szereplő utasítás azt mondja, hogy „3 képpontra a tábla bal szélétől, és 4 képpontra a tábla tetejétől". A számok mindig a vezérlőhöz (ebben az esetben a JPanel-alosztályodhoz), és nem a teljes kerethez viszonyított helyet adják meg.



Véletlenszerűen színezett kör festése fekete háttérre

```
public void paintComponent(Graphics g) {

    g.fillRect(0,0,this.getWidth(), this.getHeight());

    int red = (int) (Math.random() * 255);
    int green = (int) (Math.random() * 255);
    int blue = (int) (Math.random() * 255);

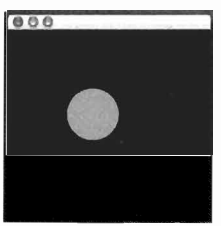
    Color randomColor = new Color(red, green, blue);
    g.setColor(randomColor);
    g.fillOval(70,70,100,100);
}
```

a teljes tábla kitöltése feketével (az alapértelmezett színnel)

Az első két argumentum a bal felső sarok x, y koordinátáját határozza meg a táblához viszonyítva. Innen indul a rajzolás, tehát a 0,0 azt jelenti, hogy „kezdd 0 képpontra a bal széltől, és 0 képpontra a felső széltől". A másik két argumentum jelentése: „Legyen ez a négyszög olyan széles, mint a tábla (this.width()), és a magassága is egyezzen meg a tábláéval (this.height)".

Szint 3 (RGB-értékeket jelképező) int átadásával keverhetsz ki.

Kezdd 70-70 képpontra a tábla bal szélétől, illetve a tetejétől, és legyen 100-100 képpont széles és magas.



Minden jó Graphics hivatkozás mögött egy Graphics2D objektum áll

A paintComponent() argumentumát Graphics típusúként vezették be (java.awt.Graphics):

```
public void paintComponent(Graphics g) { }
```

A „g" paraméter tehát EGYFAJTA (IS-A) Graphics objektum, ami azt jelenti, hogy a Graphics egy *alosztálya lehet* – és valójában *az is*.
A „g" paraméter által hivatkozott objektum valójában a Graphics2D osztály egy példánya.
De miért érdekes ez? Nos, azért, mert egy Graphics2D hivatkozással megtehetsz bizonyos dolgokat, amiket egy Graphics hivatkozással nem. Egy Graphics2D objektum többre képes egy Graphics objektumnál, és a Graphics hivatkozás mögött valójában egy Graphics2D objektum leselkedik.
Emlékezz vissza a többalakúságra! A fordító a hivatkozás, és nem az objektum típusa alapján dönti el, hogy milyen tagfüggvényeket hívhatsz meg. Ha van egy Dog objektumod, amelyre egy Animal típusú hivatkozó változó mutat...

```
Animal a = new Dog();
```

...akkor NEM mondhatod ezt:

```
a.bark();
```

Annak ellenére sem, hogy tudod, hogy valójában egy Dog-ról van szó. A fordító ránéz az „a"-ra, látja, hogy Animal típusú, és nem talál bark() gombot az Animal osztály távirányítóján. Az objektumot viszont visszaalakíthatod Dog-gá (vagyis azzá, ami *valójában*):

```
Dog d = (Dog) a;
d.bark();
```

A tanulság a Graphics objektummal kapcsolatban tehát ez:
Ha a Graphics2D osztály egy tagfüggvényét kell használnod, *nem használhatod* a paintComponent „g" paraméterét közvetlenül a tagfüggvényből, de *átalakíthatod* egy új Graphics2D típusú változóvá:

```
Graphics2D g2d = (Graphics2D) g;
```

Ezeket a tagfüggvényeket hívhatod meg egy Graphics hivatkozásra:

```
drawImage()
drawLine()
drawPolygon()
drawRect()
drawOval()
fillRect()
fillRoundRect()
setColor()
```

A Graphics2D *objektumot* így alakíthatod át Graphics2D *hivatkozássá*:

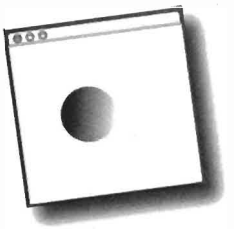
```
Graphics2D g2d = (Graphics2D) g;
```

Ezeket a tagfüggvényeket hívhatod meg egy Graphics2D hivatkozásra:

```
fill3DRect()
draw3DRect()
rotate()
scale()
shear()
transform()
setRenderingHints()
```

(itt nem soroltunk fel minden tagfüggvényt – a többit megtalálod az API-ban)

Az élet túl rövid ahhoz, hogy a kört egyszínűre fesd, amikor egy színátmenet is vár rád



```
public void paintComponent(Graphics g) {

    Graphics2D g2d = (Graphics2D) g;

    GradientPaint gradient = new GradientPaint(70,70,Color.blue, 150,150, Color.orange);

    g2d.setPaint(gradient);

    g2d.fillOval(70,70,100,100);

}
```

ez valójában egy Graphics2D objektum, ami egyszerű Graphics objektumnak álcázza magát

átalakítjuk, hogy olyasmit is meghívhassunk rá, ami a Graphics2D-nek van, de a Graphics-nak nincs

kezdőpont, kezdőszín, végpont, végszín

ez a virtuális festőecsetet egyetlen szín helyett egy színátmenetre (gradient) állítja

a fillOval() tagfüggvény azt mondja, hogy „töltsd ki az oválist azzal, amit a festőecsetbe töltöttél (vagyis a színátmenettel)"

```
public void paintComponent(Graphics g) {
    Graphics2D g2d = (Graphics2D) g;

    int red = (int) (Math.random() * 255);
    int green = (int) (Math.random() * 255);
    int blue = (int) (Math.random() * 255);
    Color startColor = new Color(red, green, blue);

    red = (int) (Math.random() * 255);
    green = (int) (Math.random() * 255);
    blue = (int) (Math.random() * 255);
    Color endColor = new Color(red, green, blue);

    GradientPaint gradient = new GradientPaint(70,70,startColor, 150,150, endColor);
    g2d.setPaint(gradient);
    g2d.fillOval(70,70,100,100);
```

Ez ugyanolyan, mint a fenti kód, csak éppen a színátmenet kezdő- és végszínét véletlenszerűen választja ki. Próbáld ki!



ESEMÉNYEK

- Ha grafikus felületet szeretnél készíteni, egy ablakkal, rendszerint egy JFrame-mel kell kezdened:
  ```
  JFrame frame = new JFrame();
  ```
- A JFrame-hez így adhatsz vezérlőket (gombokat, szövegmezőket stb.):
  ```
  frame.getContentPane().add(button);
  ```
- A legtöbb összetevőtől eltérően a JFrame nem engedi, hogy közvetlenül adj hozzá elemeket, ezért azokat a JFrame tartalomtáblájához (content pane) kell hozzáadnod.
- Az ablakot (JFrame-et) úgy jelenítheted meg, hogy megadod a méretét, és láthatóvá teszed:
  ```
  frame.setSize(300, 300);
  frame.setVisible(true);
  ```
- Ha tudni szeretnél róla, ha a felhasználó egy gombra kattintott (vagy más műveletet hajtott végre a felhasználói felületen), figyelned kell a grafikus felület megfelelő eseményeit.
- Ha egy eseményt figyelni szeretnél, jelezned kell az érdeklődésedet az esemény forrásának. Az eseményforrás az az elem (gomb, jelölőnégyzet stb.), amely a felhasználó művelete alapján „elindítja" az eseményt.
- A figyelőfelület módot ad az eseményforrásnak, hogy visszahívjon téged, azáltal, hogy meghatározza azt a tagfüggvényt (azokat a tagfüggvényeket), amelyet (amelyeket) az eseményforrás az esemény bekövetkeztekor meghívhat.
- Egy eseményre úgy iratkozhatsz fel egy forrásnál, hogy meghívod a forrás bejegyzőfüggvényét. A bejegyzőfüggvények mindig **add<*EseményTípus*>Listener** alakúak. Például ha egy gomb műveleti eseményeire (ActionEvent) szeretnél feliratkozni, ezt a hívást kell végrehajtanod:
  ```
  button.addActionListener(this);
  ```
- A figyelőfelületet a felület összes eseménykezelő tagfüggvényének megvalósításával valósíthatod meg. Az eseménykezelő kódot a figyelő visszahívó (callback) függvényében kell elhelyezned. Az ActionEvent események esetében a tagfüggvény ez:
  ```
  public void actionPerformed(ActionEvent
                                    event){
      button.setText("you clicked!");
  }
  ```
- Az eseménykezelő tagfüggvénynek átadott eseményobjektum információkat tartalmaz az eseményről, beleértve az esemény forrását.

GRAFIKA

- Kétdimenziós grafikákat közvetlenül rajzolhatsz a vezérlőkre.
- A vezérlőkön .gif és .jpeg képeket is közvetlenül megjeleníthetsz.
- Ha saját grafikát (akár egy .gif vagy .jpeg képet) szeretnél megjeleníteni, hozd létre a JPanel egy alosztályát, és írd felül a paintComponent() tagfüggvényt.
- A paintComponent() tagfüggvényt a grafikus alrendszer hívja meg, SOHA NEM TE MAGAD.
  A paintComponent() argumentuma egy Graphics objektum, amely egy felületet biztosít, amelyre rajzolhatsz, és amely végül megjelenik a képernyőn. Ezt az objektumot te magad nem hozhatod létre.
- Egy Graphics objektumra (a paintComponent paraméterére) jellemzően ilyen tagfüggvényeket szoktak meghívni:
  ```
  graphics.setColor(Color.blue);
  g.fillRect(20,50,100,120);
  ```
- Ha .jpg képet szeretnél megjeleníteni, hozz létre egy Image objektumot:
  ```
  Image image = new ImageIcon("catzil-
  la.jpg").getImage();
  ```
  Majd rajzold ki a képet így:
  ```
  g.drawImage(image,3,4,this);
  ```
- A paintComponent() Graphics típusú paramétere által hivatkozott objektum valójában a Graphics2D osztály egy példánya. A Graphics2D osztálynak különféle tagfüggvényei vannak; többek között a következők:
  ```
  fill3DRect(), draw3DRect(), rotate(),
  scale(), shear(), transform()
  ```
- A Graphics2D tagfüggvényeinek meghívásához át kell alakítanod a paramétert Graphics típusúról Graphics2D típusú objektumra:
  ```
  Graphics2D g2d = (Graphics2D) g;
  ```

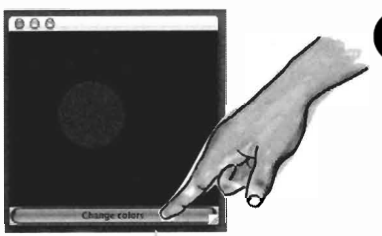
Értesülhetek egy eseményről. Rajzolhatok képeket. De rajzolhatok képet akkor is, amikor értesülök egy eseményről?

Kapcsolj egy eseményt egy a rajztáblán végrehajtani kívánt változtatáshoz: a kör változtasson színt minden alkalommal, amikor a gombra kattintasz. Íme a program menete:

Indítsd el az alkalmazást

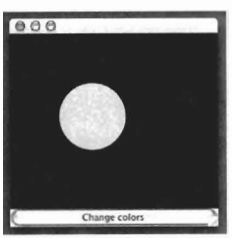


1 Felépül a keret a két vezérlővel (a rajztáblával és egy gombbal). Létrejön egy figyelő, és bejegyezteti magát a gombnál, majd megjelenik a keret, és vár, hogy a felhasználó kattintson.



2 A felhasználó a gombra kattint, a gomb pedig létrehoz egy eseményobjektumot, és meghívja a figyelő eseménykezelőjét.

3 Az eseménykezelő meghívja a repaint() függvényt a keretre. Ennek hatására a rendszer meghívja a paintComponent() tagfüggvényt a rajztáblára.



4 Voilá! A kör új színben tűnik fel, mert a paintComponent() ismét lefut, de a kört ezúttal egy véletlenszerűen kiválasztott színnel tölti ki.



Grafikus felületi elrendezések: egynél több vezérlő elhelyezése egy keretben

A grafikus felületi elrendezéseket a *következő* fejezetben tárgyaljuk, de már itt tartunk neked egy gyorstalpaló tanfolyamot, hogy továbbléphess. A kereteknek alapértelmezés szerint öt területe (region) van, amelyhez elemeket adhatsz. Egy keret minden területén csak *egy-egy* dolgot helyezhetsz el, de ne ess pánikba! Ez az egy dolog ugyanis egy panel is lehet, ami három elemet tartalmaz, köztük egy újabb panelt, amelyen két további dolog található, és így tovább. Valójában „csaltunk", amikor egy gombot így adtunk a kerethez:

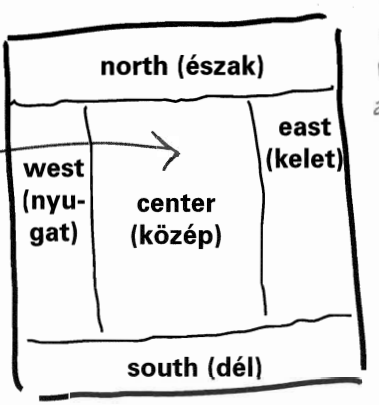
```
frame.getContentPane().add(button);
```

Ezt valójában nem így kell csinálni (az egyargumentumos add tagfüggvénnyel).

Ez a jobbik (és általában a kötelező) módja annak, hogy egy kerethez alapértelmezett tartalomtáblát adj. Mindig add meg, hogy HOL (melyik területen) szeretnéd elhelyezni a vezérlőt. Ha az egyargumentumos add tagfüggvényt hívod meg (amit nem lenne szabad használni), a vezérlő automatikusan a középső területre kerül.

```
frame.getContentPane().add(BorderLayout.CENTER, button);
```

A kétargumentumos add tagfüggvényt hívjuk meg, amelynek egy területet kell megadnod (egy állandóval), valamint azt a vezérlőt, amelyet hozzá szeretnél adni az adott területhez.



alapértelmezett terület

Hegyezd ki a ceruzád!

Az előző oldalon látott képek alapján írd meg a kódot, amely hozzáadja a gombot és a rajztáblát a kerethez.

A kör színe minden alkalommal megváltozik, amikor a gombra kattintasz

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;


public class SimpleGui3C implements ActionListener {

    JFrame frame;

    public static void main (String[] args) {
       SimpleGui3C gui = new SimpleGui3C();
       gui.go();
    }

    public void go() {
       frame = new JFrame();
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

       JButton button = new JButton("Change colors");
       button.addActionListener(this);

       MyDrawPanel drawPanel = new MyDrawPanel();

       frame.getContentPane().add(BorderLayout.SOUTH, button);
       frame.getContentPane().add(BorderLayout.CENTER, drawPanel);
       frame.setSize(300,300);
       frame.setVisible(true);
    }

    public void actionPerformed(ActionEvent event) {
        frame.repaint();
    }
}
```

Az egyedi rajztábla (egy MyDrawPanel-példány) a keret CENTER területén jelenik meg.



A gomb a keret SOUTH területén található.

Hozzáadjuk a figyelőt (this) a gombhoz.

Hozzáadjuk a két vezérlőt (a gombot és a rajztáblát) a keret két területéhez.

Amikor a felhasználó kattint, utasítjuk a keretet, hogy fesse újra (repaint()) magát. Ez azt jelenti, hogy a paintComponent() a keret minden vezérlőjére meghívódik!

```
class MyDrawPanel extends JPanel {

   public void paintComponent(Graphics g) {
      // az oválist véletlenszerű színnel kitöltő kód
      // a kódot néhány oldallal korábban találod
   }

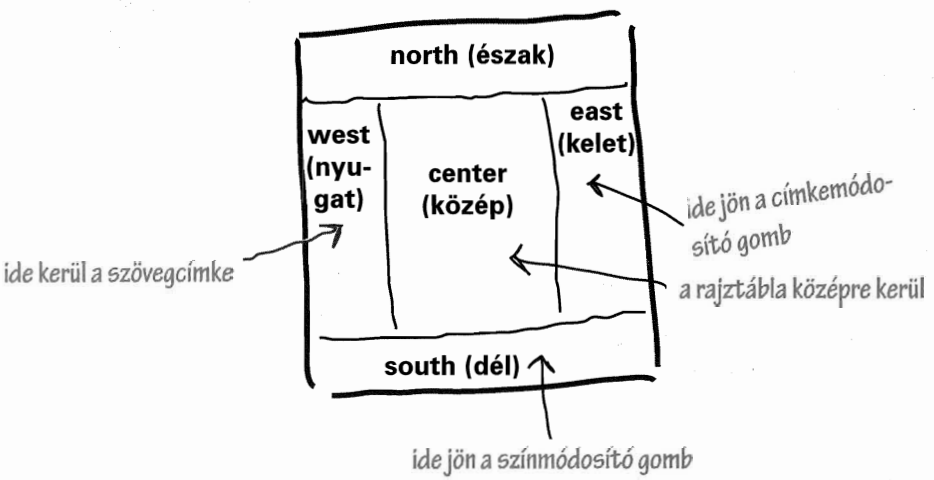
}
```

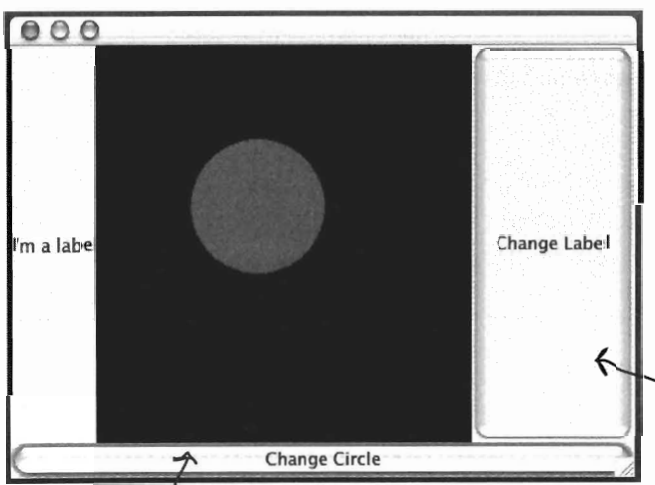
A rajztábla paintComponent() tagfüggvénye minden alkalommal meghívódik, amikor a felhasználó a gombra kattint.

Próbáld ki KÉT gombbal!

A „déli" gomb úgy fog működni, ahogy most: egyszerűen meghívja az újrafestő függvényt a keretre. A második gomb (amelyet a keleti területre ragasztunk) egy szövegcímke szövegét fogja megváltoztatni. (A címke – label – csupán egy szöveget jelenít meg a képernyőn.)

Tehát most NÉGY vezérlőre lesz szükséged





És két eseményről kell értesülnöd

Húha.
Lehetséges ez egyáltalán? Hogyan értesülhetsz *két* eseményről, amikor csak *egy* actionPerformed() tagfüggvényed van?

Hogyan értesülhetsz két különböző gomb műveleti eseményeiről, ha a két gombnak más-más dolgot kell csinálnia?

1 első lehetőség

Valósíts meg két actionPerformed() tagfüggvényt

```
class MyGui implements ActionListener {
   // itt lefut egy csomó kód, aztán:

    public void actionPerformed(ActionEvent event) {
        frame.repaint();
    }

    public void actionPerformed(ActionEvent event) {
        label.setText("That hurt!");
    }
}
```

De hát ez lehetetlen!

A hátránya: Ezt nem teheted meg! Ugyanazt a tagfüggvényt nem valósíthatod meg kétszer egy Java-osztályon belül, mert a fordító nem fogadja el. És még ha meg is *tehetnéd*, honnan tudná az eseményforrás, hogy a két tagfüggvény közül *melyiket* kell meghívnia?

2 második lehetőség

Jegyeztesd be ugyanazt a figyelőt mindkét gombbal

```
class MyGui implements ActionListener {
  // itt bevezetünk néhány példányváltozót

    public void go() {
      // felépítjük a grafikus felületet
      colorButton = new JButton();
      labelButton = new JButton();
      colorButton.addActionListener(this);
      labelButton.addActionListener(this);
      // ide további grafikus felületi kód jön...
    }

   public void actionPerformed(ActionEvent event) {
        if (event.getSource() == colorButton) {
            frame.repaint();
        } else {
           label.setText("That hurt!");
        }
    }
}
```

Jegyeztesd be ugyanazt a figyelőt mindkét gombbal.

Kérdezd meg az eseményobjektumtól, hogy melyik gomb indította el az eseményt, és ennek alapján döntsd el, mit csinálsz.

A hátránya: Ez a megoldás működik, de a legtöbb esetben nem igazán objektumközpontú. Ha egy eseménykezelő számos különböző dolgot csinál, az azt jelenti, hogy egyetlen tagfüggvénnyel különféle dolgokat végeztetsz. Ha meg kell változtatnod *egyetlen* forrás kezelésének módját, *mindegyiknek* az eseménykezelőjéhez hozzá kell nyúlnod. Ez néha *tényleg* jó megoldás, de általában rontja a karbantarthatóságot és a bővíthetőséget.

Hogyan értesülhetsz két különböző gomb műveleti eseményeiről, ha a két gombnak más-más dolgot kell csinálnia?

③ harmadik lehetőség

Hozz létre két külön ActionListener osztályt

```
class MyGui  {
   JFrame frame;
   JLabel label;
   void gui() {
     // kód, amely példányosítja a két figyelőt, és az egyiket
     // a színgombbal, a másikat pedig a címkegombbal jegyezteti be
   }
}  // az osztály lezárása
```

```
class ColorButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        frame.repaint();
    }
}
```

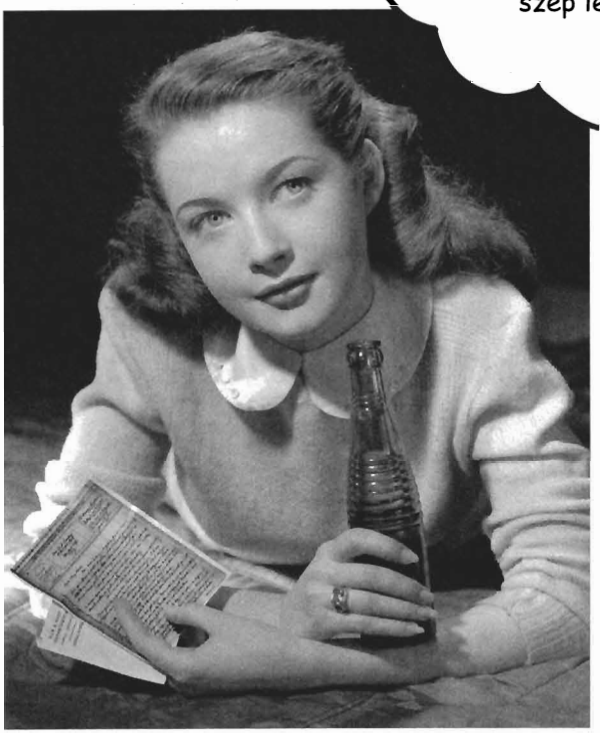
Nem működik! Ez az osztály nem rendelkezik hivatkozással a MyGui osztály „frame" változójára.

```
class LabelButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        label.setText("That hurt!");
    }
}
```

"Ez fájt!"

Gond van! Ez az osztály nem rendelkezik hivatkozással a „label" változóra.

A hátránya: Ezek az osztályok nem férnek hozzá azokhoz a változókhoz – „frame" és „label" –, amelyeken műveleteket kell végezniük. Ezt kijavíthatod, de csak úgy, ha mindkét figyelőosztálynak hivatkozást adsz a fő GUI-osztályra, hogy az actionPerformed() tagfüggvényeken belül a figyelők a GUI-osztályhivatkozás segítségével elérhessék a GUI-osztály változóit. Ez viszont megsérti az egységbe zárás (betokozás) elvét, ezért valószínűleg kiolvasó tagfüggvényeket kell írnod a grafikus felületi vezérlőkhöz (getFrame(), getLabel() stb.), és esetleg hozzá kell adnod egy konstruktort a figyelőosztályhoz, hogy a GUI-hivatkozást átadhasd a figyelőnek, amikor a figyelőből létrejön egy példány. Nos, ez bizony elég bonyolult és macerás.
Lennie kell jobb megoldásnak!



Hát nem lenne csodás, ha két különböző figyelőosztályod lenne, de ezek úgy érhetnék el a fő GUI-osztály példányváltozóit, mintha a figyelőosztályok abba az osztályba tartoznának? Így mindenből a legjobbat kapnád. Meseszép lenne. De ez csak álom...

Belső osztályok: a felmentő sereg

Megteheted, hogy egy osztályt beágyazol egy másikba, méghozzá könnyen. Csak arra kell ügyelned, hogy a belső osztály meghatározása a külső osztály kapcsos zárójelein *belül* legyen.

Egyszerű belső osztály:

```
class MyOuterClass {

    class MyInnerClass {
        void go() {
        }
    }

}
```

A belső osztályt teljesen körbefogja a külső osztály

Egy belső osztály a külső osztály valamennyi tagfüggvényét és változóját használhatja, **még a privátakat is**.

A belső osztály úgy használhatja ezeket a változókat és tagfüggvényeket, mintha azokat a belső osztály határozta volna meg.

A belső osztályok különleges engedélyt kapnak a külső osztály elemeinek használatára – *és még a privát dolgokhoz is hozzáférhetnek*. Ezenkívül a belső osztály úgy használhatja a külső osztály privát változóit és tagfüggvényeit, mintha ezeket a tagokat a belső osztály határozta volna meg. Ez az igazán kényelmes a belső osztályokban: a szokványos osztályok szinte minden előnyével rendelkeznek, de különleges hozzáférési jogaik vannak.

Külső osztályváltozót használó belső osztály:

```
class MyOuterClass {

    private int x;

    class MyInnerClass {
        void go() {
            x = 42;
        }
    } // a belső osztály lezárása

} // a külső osztály lezárása
```

Az „x" úgy használható, mintha a belső osztály változója lenne!

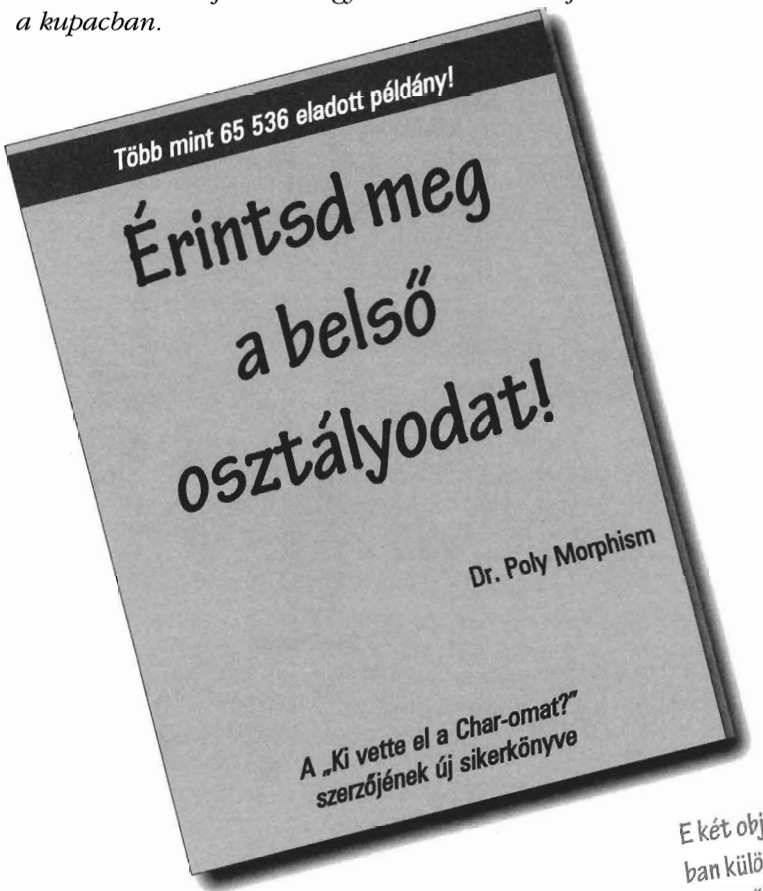
A belső osztályok példányait külső osztályok példányaihoz kell kapcsolni*

Minden belső objektum különleges kapcsolatot ápol egy külső objektummal.

Ne feledd, hogy amikor arról beszélünk, hogy egy belső *osztály* elér valamit a külső osztályban, valójában a belső osztály egy *példányáról* van szó, amely a külső osztály egy *példányában* fér hozzá valamihez. De *melyik* példányról?

A belső osztály *tetszőleges* példánya hozzáférhet a külső osztály *bármely* példányának tagfüggvényeihez és változóihoz? **Nem!**

*Minden **belső** objektumot egy konkrét **külső** objektumhoz kell kötni a kupacban.*



1. Létrehozol egy példányt a külső osztályból.



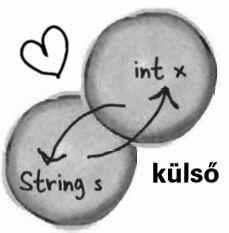
SajatKulso objektum

2. Létrehozol egy példányt a belső osztályból a külső osztály példányának felhasználásával.



SajatBelso objektum

3. A külső és a belső objektum ezzel intim kapcsolatba kerül.



E két objektum között a kupacban különleges kötelék áll fenn. A belső használhatja a külső változóit (és viszont).

*Ez alól van egy kivétel: az a különleges eset, amikor a belső osztályt egy statikus tagfüggvény belsejében határozod meg. Ebbe azonban nem megyünk bele, mert akár a teljes Java-életedet leélheted anélkül, hogy egy ilyen esettel is találkoznál.

Hogyan hozhatsz létre példányt egy belső osztályból?

Ha egy belső osztályt egy külső osztály kódjából példányosítasz, a belső objektum a külső osztály adott példányával fog „köteléket" kialakítani. Például ha egy tagfüggvény kódja hozza létre a belső osztály példányát, a belső objektum ahhoz a példányhoz fog kötődni, amelyiknek a tagfüggvénye fut.
A külső osztályokban levő kódok pontosan ugyanúgy példányosíthatják a saját belső osztályaikat, mint bármely más osztályt: **new MyInner()** (new sajatBelso()).

```
class MyOuter  {

   private int x;

   MyInner inner = new MyInner();

   public void doStuff() {
      inner.go();
    }

   class MyInner {
      void go() {
         x = 42;
       }
    } // a belső osztály lezárása

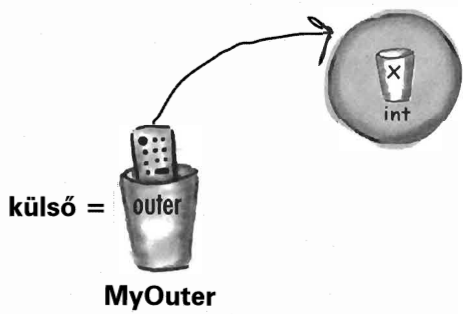
} // a külső osztály lezárása
```

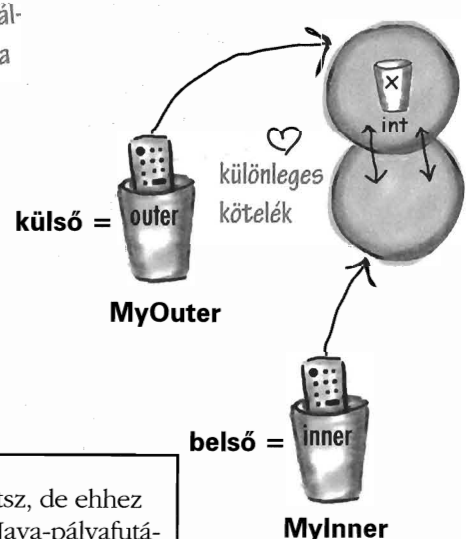
A külső osztálynak van egy „x" nevű privát példányváltozója.

Létrehozunk egy példányt a belső osztályból.

Meghívunk egy tagfüggvényt a belső osztályra.

A belső osztályban levő tagfüggvény úgy használja a külső osztály „x" példányváltozóját, mintha az a belső osztályhoz tartozna.





Oldalsáv

Belső objektumpéldányt a külső osztályon *kívül* futó kódból is létrehozhatsz, de ehhez különleges utasításformát kell használnod. Nagy rá az esély, hogy a teljes Java-pályafutásod alatt sem kell soha kívülről létrehoznod belső osztályt, de ha érdekel a dolog, íme a kód:

```
class Foo {
    public static void main (String[] args) {
       MyOuter outerObj = new MyOuter();
       MyOuter.MyInner innerObj = outerObj.new MyInner();
    }
}
```

Most már működésre bírhatod a kétgombos kódot

```
public class TwoButtons {
    JFrame frame;
    JLabel label;

    public static void main (String[] args) {
       TwoButtons gui = new TwoButtons ();
       gui.go();
    }

    public void go() {
       frame = new JFrame();
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

       JButton labelButton = new JButton("Change Label");
       labelButton.addActionListener(new LabelListener());

       JButton colorButton = new JButton("Change Circle");
       colorButton.addActionListener(new ColorListener());

       label = new JLabel("I'm a label");
       MyDrawPanel drawPanel = new MyDrawPanel();

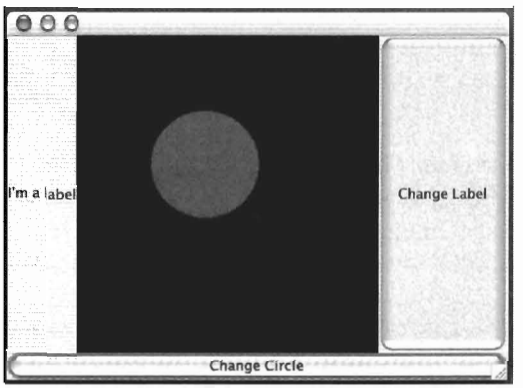
       frame.getContentPane().add(BorderLayout.SOUTH, colorButton);
       frame.getContentPane().add(BorderLayout.CENTER, drawPanel);
       frame.getContentPane().add(BorderLayout.EAST, labelButton);
       frame.getContentPane().add(BorderLayout.WEST, label);

       frame.setSize(300,300);
       frame.setVisible(true);
    }

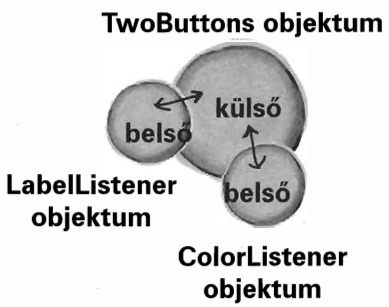
    class LabelListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            label.setText("Ouch!");
        }
    } // a belső osztály lezárása

    class ColorListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            frame.repaint();
        }
    } // a belső osztály lezárása
}
```

a fő GUI-osztály most nem valósítja meg az ActionListener felületet



ahelyett, hogy (this)-t adnánk át a gomb figyelőbejegyző tagfüggvényének, a megfelelő figyelőosztály egy új példányát adjuk át



Így KÉT ActionListener-t kapunk egyetlen osztályban!

a belső osztály tud a „label"-ről

a belső osztály anélkül használhatja a „frame" példányváltozót, hogy kifejezetten hivatkozna a külső osztály objektumára



Java – lehull a lepel

Heti interjúalanyunk: Egy Belső Osztály példánya

Agyhullám: Mi teszi a belső osztályokat nélkülözhetetlenné?

Belső objektum: Hol is kezdjem? Lehetőséget adunk rá, hogy Ön ugyanazt a felületet egy osztályban többször is megvalósíthassa. Ne feledje, hogy egy szokványos Java-osztályban nem valósíthat meg egy tagfüggvényt egynél többször. Viszont ha *belső* osztályokat használ, minden belső osztály megvalósíthatja *ugyanazt* a felületet, így *különböző* megvalósításokat írhat ugyanazokhoz a felületi tagfüggvényekhez.

Agyhullám: Miért *akarnám* kétszer megvalósítani ugyanazt a tagfüggvényt?

Belső objektum: Gondoljon a grafikus felület eseménykezelőire. Ha azt szeretné, hogy *három* gomb különbözőképpen reagáljon az eseményekre, akkor használhat *három* belső osztályt, amelyek mindegyike megvalósítja az ActionListener felületet – ami azt jelenti, hogy mindegyik osztály megvalósíthatja a saját actionPerformed tagfüggvényét.

Agyhullám: Tehát az eseménykezelés az egyetlen ok, amiért érdemes belső osztályokat használni?

Belső objektum: Ó, dehogy! Az eseménykezelők csak a legkézenfekvőbb példát szolgáltatják. Amikor csak önálló osztályra van szüksége, de azt szeretné, ha az említett osztály úgy viselkedne, mintha egy *másik* osztály része lenne, mindig egy belső osztály jelenti a legjobb – és néha az egyetlen – megoldást.

Agyhullám: Még mindig nem látok tisztán. Ha azt akarom, hogy a belső osztály úgy *viselkedjen*, mintha a külső osztály része lenne, miért írjak egyáltalán külön osztályt? Miért ne tegyen a belső osztály kódját egyenesen a külső osztályba?

Belső objektum: Az imént egy olyan példát hoztam fel, amikor egy felület több megvalósítására lehet szüksége. De még ha nem is használ felületeket, akkor is lehet, hogy két különböző *osztály* szükséges, mert a két osztály két különböző *dolgot* ábrázol. Ez a helyes objektumközpontú tervezés.

Agyhullám: Na, itt álljunk csak meg egy percre. Azt hittem, az objektumközpontú tervezés elsősorban az újrahasznosításról és a karbantartásról szól – tudja, arról az elvről, hogy ha két külön osztályunk van, akkor külön-külön, egymástól függetlenül módosíthatjuk és használhatjuk őket, ahelyett, hogy mindent egy osztályba tömködnénk, satöbbi, satöbbi. De egy *belső* osztály esetében mégis csak egyetlen *valódi* osztállyal dolgozunk, nem? A befogadó osztály az egyetlen, amely újrahasznosítható és mindenki mástól független. A belső osztályok nem igazán újrahasznosíthatók. Az igazat megvallva, már hallottam „újrahaszontalannak" is hívni őket…

Belső objektum: Nos, igaz, hogy egy belső osztály nem *annyira* könnyen újrahasznosítható, sőt néha egyáltalán nem lehet újrahasznosítani, mivel szorosan kötődik a külső osztály példányváltozóihoz és tagfüggvényeihez, de…

Agyhullám: …ez csak az én érveimet támasztja alá! Ha nem újrahasznosítható, akkor miért vesződjünk egy önálló osztállyal? Úgy értem, a felületektől eltekintve, ami egyébként nekem kerülő megoldásnak tűnik.

Belső objektum: Ahogy már említettem, gondoljon az EGYFAJTA (IS-A) kapcsolatokra és a többalakúságra.

Agyhullám: Jó, gondolok rájuk…

Belső objektum: Előfordulhat, hogy a külső és a belső osztálynak *más-más* IS-A tesztet kell teljesítenie! Kezdjük a többalakú GUI-figyelő példájával. Mi a gomb figyelőbejegyző tagfüggvényének bejelentett argumentumtípusa? Más szavakkal, mit lát, ha megnézi az API-ban, hogy milyen *dolgot* (osztály- vagy felülettípust) kell átadnia az addActionListener() tagfüggvénynek?

Agyhullám: Egy figyelőt kell átadni – valami olyasmit, ami egy bizonyos figyelőfelületet valósít meg: esetünkben az ActionListener-t. Ezzel mindenki tisztában van. Hová akar kilyukadni?

Belső objektum: Oda, hogy a többalakúságot tekintve egy olyan tagfüggvénye van, amely csak egy bizonyos *típust* fogad el: valamit, ami teljesíti az IS-A tesztet az ActionListener-re. De mi a helyzet – és ez a lényeg –, ha az Ön osztályának EGYFAJTA *osztály*típusnak, és nem felületnek kell lennie?

Agyhullám: Nem lenne célszerűbb, ha az osztálya csak *bővítené* azt az osztályt, amelynek a része szeretne lenni? Nem éppen ez a célja az alosztályok származtatásának? Ha B alosztálya A-nak, akkor B bárhol használható, ahol a program A-t vár. Tudja: „Kutyát átadni, ahol Állatot jelentettek be".

Belső objektum: Pontosan! Bingó! És mi történik, ha két különböző osztályra kell teljesítenie az IS-A tesztet? Olyan osztályokra, amelyek nem ugyanabba az öröklési hierarchiába tartoznak?

Agyhullám: Ööö, lássuk csak…Azt hiszem, kezdem érteni. Felületből mindig *megvalósíthatunk* többet, de osztályt csak *egyet bővíthetünk*. Ha *osztály*típusokról van szó, csak „egyféle EGYFAJTA" lehet.

Belső objektum: Pontosan! Egyszerre nem lehet Kutya és Gomb is. De ha Ön olyan Kutya, amelynek néha Gombnak kell lennie (hogy átadhassa magát olyan tagfüggvényeknek, amelyek Gombot várnak), a Kutya osztálynak (ami az Állat osztályt bővíti, ezért nem bővítheti a Gombot) lehet egy *belső* osztálya, amely a Gombot bővíti, és így a Kutya helyett Gombként viselkedik, vagyis ahol Gombra van szükség, a Kutya átadhatja a belső Gombját maga helyett. Másként megfogalmazva, ahelyett, hogy x.fogdGomb(this)-t mondana, a Kutya objektum x.fogdGomb(new SajatBelsoGomb())-ot mond.

Agyhullám: Tudna egy világosabb példát mondani?

Belső objektum: Emlékszik a rajztáblánkra, amikor saját alosztályt hoztunk létre a JPanel-ből? Ez az osztály jelenleg önálló, de nem belső osztály. Ez rendben is van, mert az osztálynak nincs szüksége különleges hozzáférésre a fő GUI példányváltozóihoz. De ha mégis szüksége lenne? Mi van, ha animációt jelenítünk meg a táblán, és a koordinátákat a fő alkalmazástól kapjuk (mondjuk valami olyasmi alapján, amit a felhasználó a grafikus felület egy másik részén csinál)? Ebben az esetben megtehetjük, hogy a rajztáblát belső osztállyá tesszük, így a JPanel alosztálya lehet, miközben a külső osztály továbbra is származhat bármilyen más osztályból.

Agyhullám: Már értem! A rajztábla amúgy sem eléggé újrahasznosítható ahhoz, hogy önálló osztály legyen, mivel amit kirajzol, az csak ebben az adott grafikus alkalmazásban érvényes.

Belső objektum: Na látja, csak megértette!

Agyhullám: Jó. Akkor rátérhetünk az Ön és a külső objektumpéldány közötti *kapcsolatra*.

Belső objektum: Nem értem magukat. Egy olyan komoly téma, mint a többalakúság, nem kínál elég szaftos pletykát?

Agyhullám: Magának fogalma sincs, hogy a közönség mennyit hajlandó fizetni egy kis bulvármocsokért. Szóval…valaki létrehozza Önt, és azonnal kötelék alakul ki Ön és a külső objektum között, igaz?

Belső objektum: Igen, így van. És igen, hasonlították már az előre elrendezett házasságokhoz. Nekünk nincs beleszólásunk abba, hogy melyik objektumhoz kötnek minket.

Agyhullám: Tetszik ez a házasság-hasonlat, ennél maradok. Egy belső objektum *elválhat* és újraházasodhat valaki *mással*?

Belső objektum: Nem, ez a kötelék egy életre szól.

Agyhullám: Kinek az életére? Az Önére? A külső objektuméra? Vagy mindkettőre?

Belső objektum: Az enyémre. Nem kötődhetek semmilyen más külső objektumhoz. Az egyetlen kiút számomra a szemétgyűjtés.

Agyhullám: És mi a helyzet a külső objektummal? Neki sem lehet viszonya más belső objektumokkal?

Belső objektum: Tudtam, hogy ide fog kilyukadni. Hogy *valójában* ezt akarta. Igen, igen. Az úgynevezett „társam" annyi belső objektumot tarthat, amennyit csak akar.

Agyhullám: Ettől azért még monogám? Úgy értem, sorban egymás után alakíthat ki kapcsolatokat, vagy egyszerre is lehet viszonya többekkel?

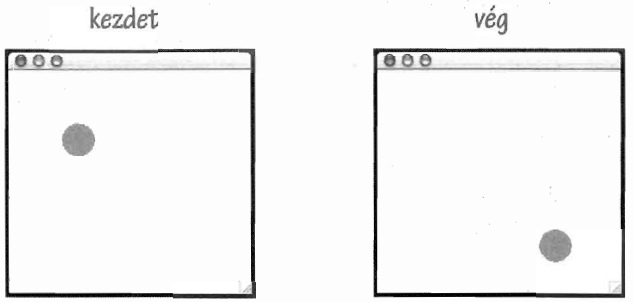
Belső objektum: Egyszerre is. Tessék. Most elégedett?

Agyhullám: Ésszerű. De ne felejtsük el, hogy *Ön* hozta fel, milyen előnyökkel jár, hogy „ugyanannak a felületnek több megvalósítása lehet". Természetes hát, hogy ha a külső osztálynak három gombja van, akkor három belső osztályra van szüksége (és így három különböző belső osztályobjektumra) az események kezeléséhez. Köszönöm az interjút. Tessék egy zsebkendő.



Belső osztály használata animációhoz

Azt már láttad, miért kényelmes belső osztályokat használni az eseményfigyelőkhöz: ugyanazt az eseménykezelő tagfüggvényt többször is megvalósíthatod. Most azonban azt nézzük meg, milyen hasznot hajthat egy belső osztály, ha valami olyasminek az alosztályaként használod, aminek a külső osztály nem bővítése: más szavakkal, amikor a külső és a belső osztály más-más öröklési fában található.
A cél egy egyszerű animáció felépítése, amelyben a kör a képernyő bal felső sarkából a jobb alsóba mozog.



Az egyszerű animáció működése

1. Rajzolunk egy objektumot egy adott x,y koordinátára:

```
g.fillOval(20,50,100,100);
```

20 képpont balról,
50 képpont felülről

2. Újrafestjük az objektumot egy másik x,y koordinátán:

```
g.fillOval(25,55,100,100);
```

25 képpont balról, 55 képpont felülről
(az objektum kicsit elmozdul jobbra lefelé)

3. Addig ismételjük az előző lépést az x és y értékek módosításával, amíg el nem érjük az animáció kívánt végpontját.

Nincsenek

hülye kérdések

K: Miért tanulunk az animációról? Kétlem, hogy játékokat fogok készíteni.

V: Lehet, hogy nem fogsz játékokat írni, de szimulációkat esetleg készítesz majd, ahol az elemek változása modellezi egy folyamat eredményét. Az is lehet, hogy olyan kijelzőeszközt kell írnod, amely mondjuk egy képet frissítve mutatja, hogy mennyi memóriát használ egy program, vagy hogy mekkora forgalom halad át a terheléselosztó kiszolgálódon. Sok olyan alkalmazás lehetséges, amelyben folyamatosan változó számok halmazát kell valami hasznossá alakítani, hogy kinyerjük a számokban rejlő információt.
Ha mindez nagyon „üzletiesen" hangzik, az csak azért van, mert ez a „hivatalos indoklás". Valójában azért tárgyaljuk itt az animációt, mert így egyszerűen tudjuk szemléltetni a belső osztályok egy másik használatát. (És persze azért, mert *szeretjük* az animációt, a következő Agyhullám-kötetünk viszont a J2EE-ről fog szólni, és *tudjuk*, hogy abba nem tudjuk belecsempészni.)

Amit valójában szeretnénk, az valahogy így néz ki:

```
class MyDrawPanel extends JPanel {
      public void paintComponent(Graphics g) {
         g.setColor(Color.orange);
         g.fillOval(x,y,100,100);
      }
}
```



Az ovális a paintComponent() minden meghívásakor más helyen rajzolódik ki.



Hegyezd ki a ceruzád!

De honnan vesszük az új x és y koordinátákat?
És ki hívja meg a repaint() tagfüggvényt?

Próbálj **kidolgozni egy egyszerű megoldást** arra, hogy a labda a rajztábla bal felső sarkából a jobb alsóba mozogjon. A mi megoldásunk a következő oldalon található: addig ne lapozz, amíg kész nem vagy!

Óriási Segítség: tedd a rajztáblát belső osztállyá.
Másik Segítség: ne tegyél semmilyen ismétlődő ciklust a paintComponent() tagfüggvénybe.

Írd ide az ötleteidet (vagy a kódot):

Az egyszerű animáció teljes kódja

```
import javax.swing.*;
import java.awt.*;


public class SimpleAnimation {

    int x = 70;
    int y = 70;
```

Létrehozunk két példányváltozót a fő GUI-osztályban a kör x és y koordinátái számára.

```
    public static void main (String[] args) {
       SimpleAnimation gui = new SimpleAnimation ();
       gui.go();
    }

    public void go() {
       JFrame frame = new JFrame();
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

       MyDrawPanel drawPanel = new MyDrawPanel();

       frame.getContentPane().add(drawPanel);
       frame.setSize(300,300);
       frame.setVisible(true);
```

Itt nincs semmi újdonság. Létrehozzuk a vezérlőket, és elhelyezzük azokat a keretben.

Itt történik a lényeg!

```
       for (int i = 0; i < 130; i++) {

          x++;
          y++;
```

ezt 130-szor ismételjük

növeljük az x és y koordinátákat

```
          drawPanel.repaint();
```

utasítjuk a táblát, hogy fesse újra magát (hogy a kört az új helyén lássuk)

```
          try {
            Thread.sleep(50);
          } catch(Exception ex) { }
       }

    } // a go() tagfüggvény lezárása
```

Kicsit lelassítjuk (különben olyan gyorsan fog mozogni, hogy nem LÁTOD a mozgását). Ne aggódj, ezt még nem kellett tudnod. A szálakhoz a 15. fejezetben jutunk majd el.

Ez most egy belső osztály.

```
    class MyDrawPanel extends JPanel {

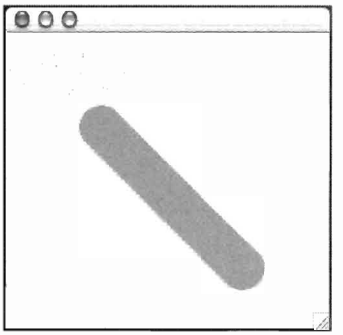
       public void paintComponent(Graphics g) {
          g.setColor(Color.green);
          g.fillOval(x,y,40,40);
```

A külső osztály folyamatosan frissített x és y koordinátáit használjuk.

```
       }
    } // a belső osztály lezárása
} // a külső osztály lezárása
```

Hm. Nem mozog... csak elkenődik.

Mit csináltunk rosszul?
A paintComponent() tagfüggvényben van egy kis hiba.



Nem pont ezt akartuk.

Elfelejtettük törölni, amit korábban rajzoltunk, ezért húzunk csíkot.

A hiba kijavításához csak annyit kell tenned, hogy minden alkalommal kitöltöd az egész táblát a háttérszínnel, mielőtt kirajzolnád a kört. Az alábbi kód két sort ad a tagfüggvény elejéhez: az egyik fehérre (white) állítja a színt (ez a rajztábla háttérszíne), míg a másik kitölti a teljes tábla négyszögét ezzel a színnel. Magyarul, a kód azt mondja, hogy „töltsd ki a 0,0 x-y koordinátán kezdődő (vagyis balról és felülről egyaránt 0 képpontra levő) négyszöget, ami legyen olyan széles és magas, mint jelenleg a tábla".

```
public void paintComponent(Graphics g) {
   g.setColor(Color.white);
   g.fillRect(0,0,this.getWidth(), this.getHeight());

   g.setColor(Color.green);
   g.fillOval(x,y,40,40);

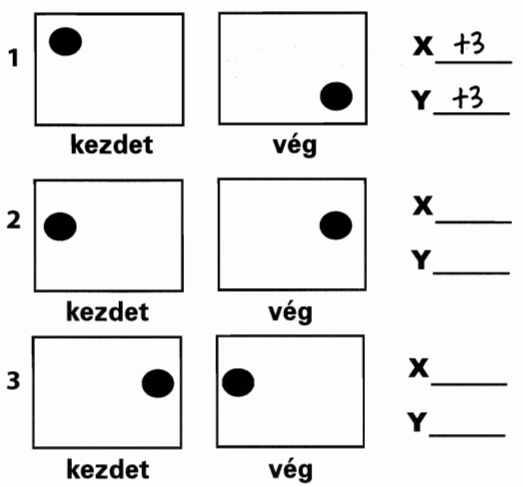
}
```

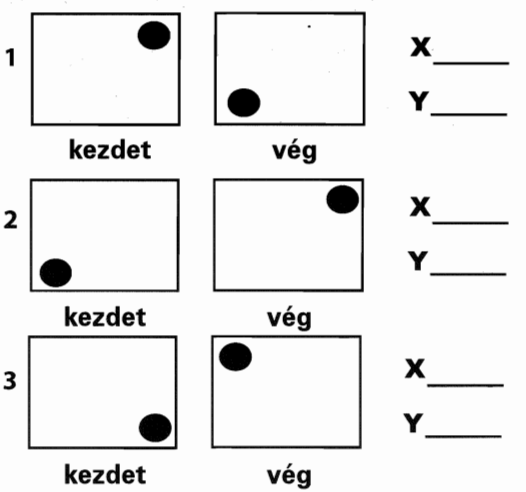
a getWidth() és getHeight() tagfüggvényeket a JPanel-től örököltük



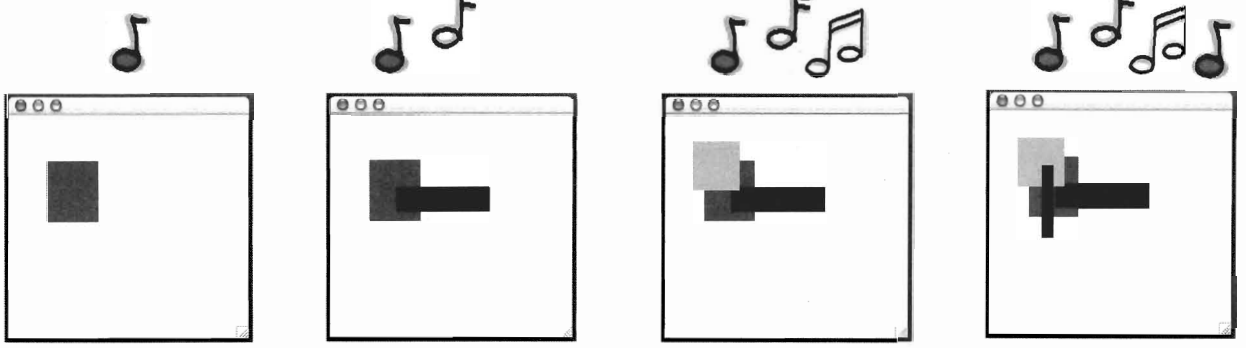
Hegyezd ki a ceruzád!

Hogyan változtatnád az x és y koordinátákat, hogy az alábbi animációk álljanak elő belőlük? (Az első példában a kör 3 képpontos lépésközökkel mozog.)





Kódkonyha



első ütem | második ütem | harmadik ütem | negyedik ütem

Készíts zenés videót a Java által véletlenszerűen előállított rajzokból, amelyek a zene üteméhez igazodnak!
Közben egy újfajta, nem grafikus felületi eseményt is figyelsz majd, amelyet maga a zene indít el.

Ne feledd, ez a rész nem kötelező – de szerintünk hasznodra válhat, és tetszeni fog.
És persze lenyűgözhetsz vele másokat.
(Na jó, lehet, hogy csak azokra hat, akiket tényleg könnyű lenyűgözni, de akkor is...)

Nem grafikus felületi események figyelése

Na jó, talán nem igazi zenés videó lesz, de akkor is egy olyan program, amely véletlenszerű képeket rajzol a képernyőre a zene ütemére. Dióhéjban: a program figyeli a zenét, és minden ütemre rajzol egy véletlenszerű négyszöget.

Ez persze új kérdéseket vet fel. Eddig csak a grafikus felület eseményeit figyelted, most viszont egy bizonyos fajta MIDI-eseményről kell értesülnöd. Szerencsére a nem grafikus felületi eseményeket ugyanúgy kezelheted, mint a GUI-eseményeket: megvalósítasz egy figyelőfelületet, bejegyezteted a figyelőt egy eseményforrással, majd hátradőlsz, és várod, hogy az eseményforrás meghívja az eseménykezelő tagfüggvényedet (amelyet a figyelőfelület határoz meg).

A legegyszerűbben úgy figyelhetnéd a zene ütemét, hogy értesítést kérsz a MIDI-eseményekről, így amikor az esemény eljut a sorozat-feldolgozóhoz, a kódod is megkapja, és megrajzolhatja a grafikát. Csakhogy van egy probléma – konkrétan egy hiba, ami nem engedi, hogy figyeld az *általad létrehozott* MIDI-eseményeket (amelyeket a NOTE ON-hoz adsz meg).

Egy apró kerülőútra lesz szükség. Van egy másik MIDI-eseménytípus, amit viszont figyelhetsz: a ControllerEvent (vezérlőesemény). A megoldás az, hogy feliratkozol a ControllerEvent eseményekre, majd gondoskodsz róla, hogy minden NOTE ON eseménnyel együtt, ugyanarra az „ütemre", elinduljon egy megfelelő ControllerEvent is. De hogyan érheted el, hogy a ControllerEvent ugyanakkor induljon el? Hát úgy, hogy hozzáadod a sávhoz – ugyanúgy, mint bármely más eseményt. Más szavakkal, a zenei utasítássorozatunk így fest:

1. ütem – NOTE ON, CONTROLLER EVENT
2. ütem – NOTE OFF
3. ütem – NOTE ON, CONTROLLER EVENT
4. ütem – NOTE OFF

és így tovább.

Mielőtt azonban elmerülnénk a teljes program tárgyalásában, tegyük egy kicsit egyszerűbbé a MIDI-üzenetek és -események létrehozását és hozzáadását, mert *ebben* a programban nagyon sokszor lesz rájuk szükség.

Ezt kell tennie a zenére rajzoló programnak:

1. MIDI-üzenetek/események sorozatát kell létrehoznia, hogy véletlenszerű hangokat játsszon le a zongorán (vagy amilyen hangszert választasz).
2. Be kell jegyeztetnie egy figyelőt az eseményekhez.
3. El kell indítania a sorozat-feldolgozót a lejátszáshoz.
4. A figyelő eseménykezelő tagfüggvényének minden meghívásakor rajzolnia kell egy véletlenszerű négyszöget a rajztáblára, és meg kell hívnia a repaint függvényt.

A programot három lépésben építjük fel:

1. Első változat: A kód leegyszerűsíti a MIDI-események létrehozását és hozzáadását, mivel nagyon sok lesz belőlük.
2. Második változat: Feliratkozunk az események figyelésére, de még nem rajzolunk. A program minden ütemre egy üzenetet ír ki a parancssorban.
4. Harmadik változat: Az igazi. A második változatot kiegészíti a rajzolással.

Egyszerűbb mód az üzenetek / események létrehozásához

Jelenleg nagyon fárasztó üzeneteket és eseményeket létrehozni, és hozzáadni egy sávhoz. Minden üzenet esetében létre kell hoznod az üzenetpéldányt (itt egy ShortMessage-példányt), meg kell hívnod a setMessage() tagfüggvényt, létre kell hoznod egy MidiEvent-et az üzenethez, és hozzá kell adnod az eseményt a sávhoz.
Az előző fejezet kódjában minden üzenetnél elvégeztük ezeket a lépéseket – ami nyolc kódsort jelentett egyetlen hang lejátszásához és a lejátszás megállításához: négy sor kellett a NOTE ON esemény hozzáadásához, és négy a NOTE OFF eseményhez:

```
ShortMessage a = new ShortMessage();
a.setMessage(144, 1, note, 100);
MidiEvent noteOn = new MidiEvent(a, 1);
track.add(noteOn);

ShortMessage b = new ShortMessage();
b.setMessage(128, 1, note, 100);
MidiEvent noteOff = new MidiEvent(b, 16);
track.add(noteOff);
```

Ezeknek a dolgoknak kell megtörténniük minden esemény esetében:

1. Egy üzenetpéldány létrehozása:
```
ShortMessage first = new ShortMessage();
```

2. A setMessage() meghívása az utasításokkal:
```
first.setMessage(192, 1, hangszer, 0)
```

3. MidiEvent-példány létrehozása az üzenethez:
```
MidiEvent noteOn = nw MidiEvent(first, 1);
```

4. Az esemény hozzáadása a sávhoz:
```
track.add(noteOn);
```

Írj egy statikus segédfüggvényt, ami létrehoz egy üzenetet, és egy MidiEvent-et ad vissza!

A „tick" esemény azt jelzi, hogy MIKOR kell végrehajtani ennek az üzenetnek az utasításait

az üzenet négy argumentuma

```
public static MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
    MidiEvent event = null;
    try {
       ShortMessage a = new ShortMessage();
       a.setMessage(comd, chan, one, two);
       event = new MidiEvent(a, tick);

    }catch(Exception e) { }
    return event;
}
```

Fú! Ennek a tagfüggvénynek öt paramétere van!

az üzenet és az esemény létrehozása a tagfüggvény paraméterei alapján

az esemény (egy az üzenettel feltöltött MidiEvent) visszaadása

Példa: az új, statikus makeEvent() tagfüggvény használata

Itt még nincs eseménykezelés vagy rajzolás, csak 15 hang sorozata, amelyek felkúsznak a skálán. Ez a kód csupán azt a célt szolgálja, hogy megtanuld az új makeEvent() tagfüggvény használatát. Ennek a tagfüggvénynek köszönhetően a következő két változat kódja sokkal rövidebb és egyszerűbb lesz.

```
import javax.sound.midi.*;
public class MiniMusicPlayer1 {

   public static void main(String[] args) {

      try {

         Sequencer sequencer = MidiSystem.getSequencer();
         sequencer.open();


         Sequence seq = new Sequence(Sequence.PPQ, 4);
         Track track = seq.createTrack();

      for (int i = 5; i < 61; i+= 4) {

         track.add(makeEvent(144,1,i,100,i));
         track.add(makeEvent(128,1,i,100,i + 2));

       } // a ciklus vége

         sequencer.setSequence(seq);
         sequencer.setTempoInBPM(220);
         sequencer.start();
      } catch (Exception ex) {ex.printStackTrace();}
   } // a main lezárása

   public static MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
         MidiEvent event = null;
         try {
           ShortMessage a = new ShortMessage();
           a.setMessage(comd, chan, one, two);
          event = new MidiEvent(a, tick);

         }catch(Exception e) { }
        return event;
      }
} // az osztály lezárása
```

ne feledkezz meg a betöltésről

létrehozunk (és megnyitunk) egy sorozat-feldolgozót

létrehozunk egy utasítássorozatot és egy sávot

létrehozunk egy csomó eseményt a skálázáshoz (a zongora 5. hangjától a 61.-ig)

elindítjuk a feldolgozást

meghívjuk az új makeEvent() tagfüggvényünket az üzenet és az esemény létrehozásához, majd az eredményt (a makeEvent() által visszaadott MidiEvent-et) hozzáadjuk a sávhoz – ezek NOTE ON(144) és NOTE OFF(128) párok

Második változat: értesülés a vezérlőeseményekről

A ControllerEvent eseményeket kell figyelnünk, ezért ezt a figyelőfelületet valósítjuk meg.

```
import javax.sound.midi.*;
public class MiniMusicPlayer2 implements ControllerEventListener {

    public static void main(String[] args) {
        MiniMusicPlayer2 mini = new MiniMusicPlayer2();
        mini.go();
    }
  public void go() {

     try {
        Sequencer sequencer = MidiSystem.getSequencer();
        sequencer.open();

        int[] eventsIWant = {127};
        sequencer.addControllerEventListener(this, eventsIWant);

        Sequence seq = new Sequence(Sequence.PPQ, 4);
        Track track = seq.createTrack();

        for (int i = 5; i < 60; i+= 4) {
           track.add(makeEvent(144,1,i,100,i));

           track.add(makeEvent(176,1,127,0,i));

           track.add(makeEvent(128,1,i,100,i + 2));
        } // a ciklus vége

        sequencer.setSequence(seq);
        sequencer.setTempoInBPM(220);
        sequencer.start();
     } catch (Exception ex) {ex.printStackTrace();}
  } // lezárás

  public void controlChange(ShortMessage event) {
        System.out.println("la");
  }

  public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
        MidiEvent event = null;
        try {
          ShortMessage a = new ShortMessage();
          a.setMessage(comd, chan, one, two);
          event = new MidiEvent(a, tick);

        }catch(Exception e) { }
        return event;
      }
} // az osztály lezárása
```

Feliratkozunk az eseményekre a sorozat-feldolgozónál. A bejegyzőfüggvénynek a figyelőt ÉS egy int tömböt kell átadni, amely a kívánt vezérlőesemények listáját jelképezi. Mi csak egy eseményre – a 127-es számúra – vagyunk kíváncsiak.

Így vesszük fel a ritmust – beszúrjuk a SAJÁT vezérlőeseményünket (a 176 azt mondja, hogy az esemény típusa ControllerEvent) a 127-es eseményszámot tartalmazó argumentummal. Ez az esemény nem csinál SEMMIT! CSAK azért tesszük ide, hogy minden alkalommal kapjunk egy eseményt, amikor a program lejátszik egy hangot. Más szavakkal, az egyetlen célja az, hogy elinduljon valami, amit MI figyelhetünk (a NOTE ON/OFF eseményeket nem figyelhetjük). Vedd észre, hogy ezt az eseményt UGYANARRA az ütemre (tick) indítjuk el, mint amikor a NOTE ON bekövetkezik. Tehát amikor NOTE ON eseményre kerül sor, tudni fogunk róla, mert ugyanakkor a MI eseményünk is elindul.

Az eseménykezelő tagfüggvény (a ControllerEvent figyelőfelületből). Minden alkalommal, amikor bekövetkezik az esemény, a parancssorba kiírjuk, hogy „la".

Az előző változattól eltérő kódrészeket szürkével emeltük ki (és ezúttal nem futtatjuk az egészet a main()-en belül).

Harmadik változat: rajzolás a zene ütemére

Ez a végső változat a második változatra épül, de hozzáadja a grafikus felülettel kapcsolatos részeket. Felépítünk egy keretet, hozzáadunk egy rajztáblát, és minden alkalommal, amikor bekövetkezik egy esemény, rajzolunk egy új négyszöget, és újrafestjük a képernyőt. A második változathoz képest az egyetlen változás, hogy a hangok véletlenszerűen szólalnak meg, nem pedig felfelé haladnak a skálán.
A kódban történt legfontosabb változás (az egyszerű grafikus felület felépítésén kívül) az, hogy a rajztáblával valósíttatjuk meg a ControllerEventListener felületet, nem pedig magával a programmal. Így amikor a rajztábla (ami egy belső osztály) értesül az eseményről, tudni fogja, hogy rajzolnia kell egy négyszöget.
A végső változat teljes kódja a következő oldalon található.

A rajztábla belső osztálya:

A rajztábla egy figyelő.

```
class MyDrawPanel extends JPanel implements ControllerEventListener {

    boolean msg = false;

    public void controlChange(ShortMessage event) {
       msg = true;
       repaint();
    }

    public void paintComponent(Graphics g) {
     if (msg) {

       Graphics2D g2 = (Graphics2D) g;

       int r = (int) (Math.random() * 250);
       int gr = (int) (Math.random() * 250);
       int b = (int) (Math.random() * 250);

       g.setColor(new Color(r,gr,b));

       int ht = (int) ((Math.random() * 120) + 10);
       int width = (int) ((Math.random() * 120) + 10);
       int x = (int) ((Math.random() * 40) + 10);
       int y = (int) ((Math.random() * 40) + 10);
       g.fillRect(x,y,ht, width);
       msg = false;

     } // az if lezárása
   } // a tagfüggvény lezárása
} // a belső osztály lezárása
```

Egy jelzőt false értékre állítunk, és csak akkor változtatjuk true-ra, ha eseményről értesülünk.

Esemény történt, ezért a jelzőt true-ra állítjuk, és meghívjuk a repaint()-et.

Azért kell jelzőt használnunk, mert MÁS dolgok is újrafestést válthatnak ki, és mi CSAK AKKOR szeretnénk rajzolni, amikor ControllerEvent következik be.

A kód többi része előállít egy véletlenszerű színt, és rajzol egy „majdnem véletlenszerű" négyszöget.



Hegyezd ki a ceruzád!

Ez a harmadik változat teljes kódja, amely közvetlenül a második változatra épül. Próbáld magad ellátni megjegyzésekkel, anélkül, hogy visszalapoznál az előző oldalakra.

```
import javax.sound.midi.*;
import java.io.*;
import javax.swing.*;
import java.awt.*;

public class MiniMusicPlayer3 {

     static JFrame f = new JFrame("My First Music Video");
     static MyDrawPanel ml;

     public static void main(String[] args) {
           MiniMusicPlayer3 mini = new MiniMusicPlayer3();
           mini.go();
     }  // a tagfüggvény lezárása


     public  void setUpGui() {
        ml = new MyDrawPanel();
        f.setContentPane(ml);
        f.setBounds(30,30, 300,300);
        f.setVisible(true);
    }  // a tagfüggvény lezárása

    public void go() {
       setUpGui();

       try {

          Sequencer sequencer = MidiSystem.getSequencer();
          sequencer.open();
          sequencer.addControllerEventListener(ml, new int[] {127});
          Sequence seq = new Sequence(Sequence.PPQ, 4);
          Track track = seq.createTrack();

          int r = 0;
          for (int i = 0; i < 60; i+= 4) {

             r = (int) ((Math.random() * 50) + 1);
             track.add(makeEvent(144,1,r,100,i));
             track.add(makeEvent(176,1,127,0,i));
             track.add(makeEvent(128,1,r,100,i + 2));
          }  // a ciklus lezárása

          sequencer.setSequence(seq);
          sequencer.start();
          sequencer.setTempoInBPM(120);
       } catch (Exception ex) {ex.printStackTrace();}
    }  // a tagfüggvény lezárása
```

```
  public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
      MidiEvent event = null;
      try {
         ShortMessage a = new ShortMessage();
         a.setMessage(comd, chan, one, two);
          event = new MidiEvent(a, tick);

       }catch(Exception e) { }
       return event;
    }   // a tagfüggvény lezárása


class MyDrawPanel extends JPanel implements ControllerEventListener {
     boolean msg = false;

     public void controlChange(ShortMessage event) {
        msg = true;
        repaint();
     }

     public void paintComponent(Graphics g) {
      if (msg) {

        Graphics2D g2 = (Graphics2D) g;

        int r = (int) (Math.random() * 250);
        int gr = (int) (Math.random() * 250);
        int b = (int) (Math.random() * 250);

        g.setColor(new Color(r,gr,b));

        int ht = (int) ((Math.random() * 120) + 10);
        int width = (int) ((Math.random() * 120) + 10);

        int x = (int) ((Math.random() * 40) + 10);
        int y = (int) ((Math.random() * 40) + 10);

        g.fillRect(x,y,ht, width);
        msg = false;

      }   // az if lezárása
    }   // a tagfüggvény lezárása
  }   // a belső osztály lezárása

}   // az osztály lezárása
```



Néhány Java-nagymenő kosztümbe öltözve álarcosbált tart. Adnak neked egy kis segítséget, és ki kell találnod, hogy ki kicsoda. Tegyük fel, hogy mindig igazat állítanak magukról. Ha olyasmit találnak mondani, ami többükre is jellemző lehet, válaszd ki mindegyiket, akire ráillik a leírás, és írd a nevüket az adott állítás mellé.

A mai vendégek:

Ebből a fejezetből bármelyik elbűvölő személyiség feltűnhet!

Az egész grafikus felület a kezemben van. _____

Minden eseménytípushoz tartozik egy belőlem. _____

Én vagyok a figyelő legfontosabb tagfüggvénye. _____

Én vagyok az a tagfüggvény, aki megadja egy JFrame-nek a méretét. _____

Olyan tagfüggvény vagyok, akihez csak hozzáadhatsz kódot, de soha nem hívhatsz meg magad. _____

Akkor jövök én, amikor a felhasználó csinál valamit. _____

A többségünk eseményforrás. _____

Én szállítom az adatokat a figyelőnek. _____

Ha én vagyok Xxx objektum, akkor az addXxxListener() tagfüggvény azt mondja, hogy én egy _____ vagyok. _____

Velem iratkoznak fel a figyelők. _____

Én vagyok az a tagfüggvény, aki az egész rajzolókódot tartalmazza. _____

Jellemzően egy objektumpéldányhoz kötődöm. _____

A (Graphics g)-ben a „g" valójában az én osztályomba tartozik. _____

Én vagyok az a tagfüggvény, aki elindítja a paintComponent()-et. _____

Én vagyok az a csomag, amelyben a legtöbb „Swinger" lakik. _____



Gyakorlat

Légy a fordító!



Az ezen az oldalon található Java-fájl egy teljes forrásfájl. A te feladatod az, hogy eljátszd a fordítóprogram szerepét, és kideríts, hogy lefordítható-e ez a fájl. Ha nem, hogyan javítanád ki? Ha igen, mit csinál?

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

class InnerButton {

  JFrame frame;
  JButton b;

  public static void main(String [] args) {
    InnerButton gui = new InnerButton();
    gui.go();
  }

  public void go() {
    frame = new JFrame();
    frame.setDefaultCloseOperation(
                  JFrame.EXIT_ON_CLOSE);

    b = new JButton("A");
    b.addActionListener();

    frame.getContentPane().add(
                  BorderLayout.SOUTH, b);
    frame.setSize(200,100);
    frame.setVisible(true);
  }

  class BListener extends ActionListener {
    public void actionPerformed(ActionEvent e) {
      if (b.getText().equals("A")) {
        b.setText("B");
      } else {
        b.setText("A");
      }
    }
  }
}
```



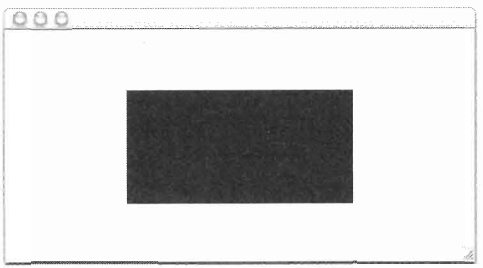
Halászd ki a medencéből!



A feladatod az, hogy kihalászd a kóddarabokat a medencéből, és elhelyezd azokat a kód üresen hagyott helyein. Egy kóddarabot egynél többször is felhasználhatsz, viszont nem lesz szükséged minden darabra. A cél egy olyan osztály felépítése, amely lefordítható, futtatható, és az alább látható kimenetet eredményezi.

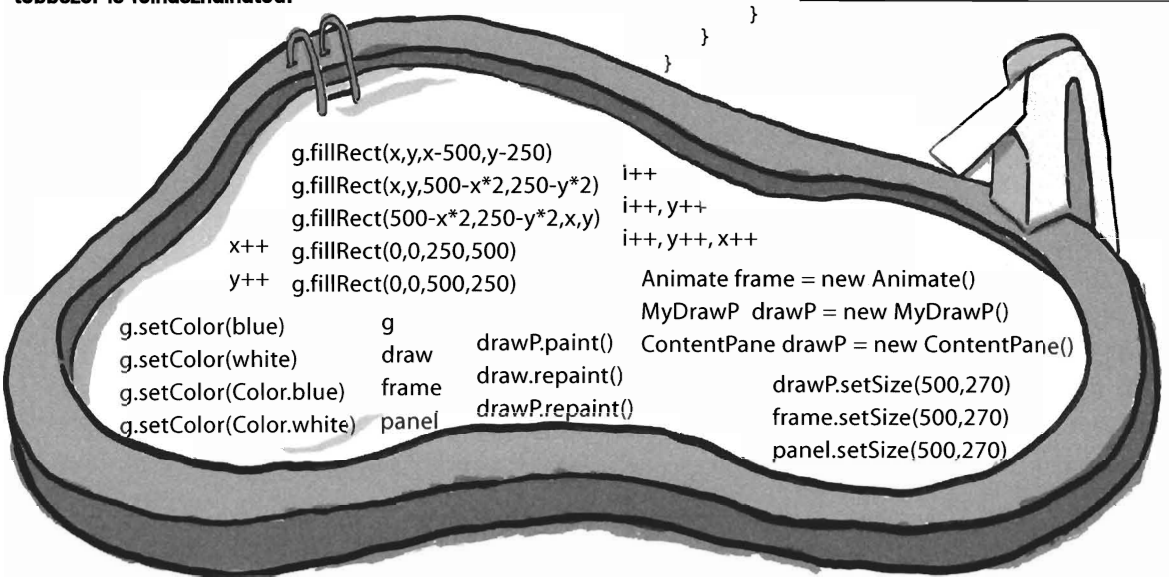
Kimenet

A Csodálatos Zsugorodó Kék Négyszög. Ez a program egy kék négyszöget eredményez, amely addig zsugorodik, amíg el nem tűnik a fehér mezőben.



```
import javax.swing.*;
import java.awt.*;
public class Animate {
    int x = 1;
    int y = 1;
    public static void main (String[] args) {
       Animate gui = new Animate ();
       gui.go();
    }
    public void go() {
       JFrame _____ = new JFrame();
       frame.setDefaultCloseOperation(
                     JFrame.EXIT_ON_CLOSE);
       _____;
       _____.getContentPane().add(drawP);
       _____;
       _____.setVisible(true);
       for (int i=0; i<124; _____) {
       _____;
       _____;
       try {
          Thread.sleep(50);
       } catch(Exception ex) { }
       }
    }
    class MyDrawP extends JPanel {
       public void paintComponent (Graphic
                                   _____) {
          _____;
          _____;
          _____;
          _____;
       }
    }
}
```

Megjegyzés: A medencében úszó kóddarabok mindegyikét többször is felhasználhatod!





Gyakorlat – Megoldás

Álarcosbál

Az egész grafikus felület a kezemben van. *JFrame*

Minden eseménytípushoz tartozik egy belőlem. *figyelőfelület*

Én vagyok a figyelő legfontosabb tagfüggvénye. *actionPerformed()*

Én vagyok az a tagfüggvény, aki megadja egy JFrame-nek a méretét. *setSize()*

Olyan tagfüggvény vagyok, akihez csak hozzáadhatsz kódot, de soha nem hívhatsz meg magad. *paintComponent()*

Akkor jövök én, amikor a felhasználó csinál valamit. *esemény*

A többségünk eseményforrás. *Swing-összetevők*

Én szállítom az adatokat a figyelőnek. *eseményobjektum*

Ha én vagyok Xxx objektum, akkor az addXxxListener() tagfüggvény azt mondja, hogy én egy _____ vagyok. *eseményforrás*

Velem iratkoznak fel a figyelők. *addActionListener()*

Én vagyok az a tagfüggvény, aki az egész rajzolókódot tartalmazza. *paintComponent()*

Jellemzően egy objektumpéldányhoz kötődöm. *belső osztály*

A (Graphics g)-ben a „g" valójában az én osztályomba tartozik. *Graphics2d*

Én vagyok az a tagfüggvény, aki elindítja a paintComponent()-et. *repaint()*

Én vagyok az a csomag, amelyben a legtöbb „Swinger" lakik. *javax.swing*

Légy a fordító!

Miután kijavítottad, ez a kód létrehoz egy grafikus felületet egy gombbal, amely kattintásra vált A és B között.

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

class InnerButton {

  JFrame frame;
  JButton b;

  public static void main(String [] args) {
    InnerButton gui = new InnerButton();
    gui.go();
  }

  public void go() {
    frame = new JFrame();
    frame.setDefaultCloseOperation(
                    JFrame.EXIT_ON_CLOSE);

    b = new JButton("A");
    b.addActionListener( new BListener() );

    frame.getContentPane().add(
                    BorderLayout.SOUTH, b);
    frame.setSize(200,100);
    frame.setVisible(true);
  }

  class BListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if (b.getText().equals("A")) {
        b.setText("B");
      } else {
        b.setText("A");
      }
    }
  }
}
```

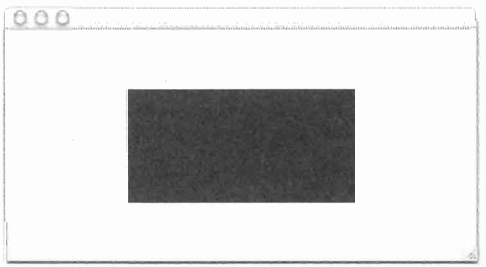
Az addActionListener() tagfüggvény egy osztályt vár, amely megvalósítja az ActionListener felületet.

Az ActionListener egy felület, a felületeket pedig nem bővítjük, hanem megvalósítjuk.



Halászd ki a medencéből!

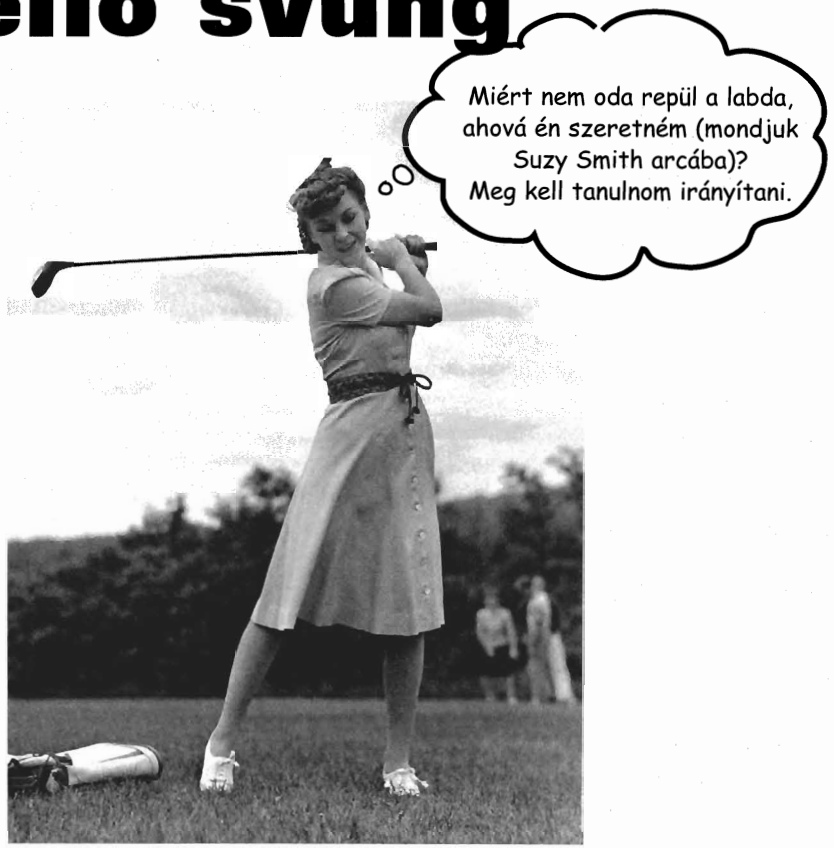
A Csodálatos Zsugorodó Kék Négyszög.



```
import javax.swing.*;
import java.awt.*;
public class Animate {
    int x = 1;
    int y = 1;
    public static void main (String[] args) {
       Animate gui = new Animate ();
       gui.go();
   }
   public void go() {
       JFrame frame = new JFrame();
       frame.setDefaultCloseOperation(
                        JFrame.EXIT_ON_CLOSE);
       MyDrawP drawP = new MyDrawP();
       frame.getContentPane().add(drawP);
       frame.setSize(500,270);
       frame.setVisible(true);
       for (int i = 0; i < 124; i++,x++,y++ ) {
          x++;
          drawP.repaint();
          try {
            Thread.sleep(50);
          } catch(Exception ex) { }
       }
   }
   class MyDrawP extends JPanel {
       public void paintComponent(Graphics g ) {
          g.setColor(Color.white);
          g.fillRect(0,0,500,250);
          g.setColor(Color.blue);
          g.fillRect(x,y,500-x*2,250-y*2);
       }
   }
}
```

13 A Swing használata

A kellő svung



A Swing-gel suhintani egyszerű. Kivéve, ha tényleg *érdekel*, hol kötnek ki a dolgok a képernyőn. A Swing-kód egyszerűnek *tűnik*, de amikor lefordítod és futtatod, rápillantasz az eredményre, és azt mondod: „Hé, *annak* nem *oda* kellett volna kerülnie!". Ami *megkönnyíti* a *kódolást*, az ugyanaz a dolog, mint ami *nehezen irányíthatóvá* is teszi azt: az **elrendezéskezelő**. Az elrendezéskezelő (Layout Manager) objektumok szabályozzák a vezérlők méretét és helyét a Java grafikus felhasználói felületein. Rengeteg munkát elvégeznek helyetted, de az eredmény nem mindig fog tetszeni. Azt akarod, hogy két gomb egyforma méretű legyen, de nem azok. Azt szeretnéd, hogy egy szövegmező három hüvelyk hosszú legyen, de kilenc. Vagy egy. És a szövegcímke *alatt* jelenik meg, és nem *mellette*. Egy kis munkával azonban engedelmességre bírhatod az elrendezéskezelőket. Ebben a fejezetben a Swing használatát gyakorolhatod, és az elrendezéskezelőkön kívül a vezérlőkkel kapcsolatos ismereteidet is bővítheted: létrehozod és megjeleníted őket (ott, ahol *te* akarod), és felhasználod őket egy programban. Suzy-nak nem túl jók a kilátásai.

Swing-összetevők

Az ***összetevő*** (component) a pontosabb kifejezés arra, amit eddig *vezérlőnek* (kütyünek, widget) hívtunk. Az összetevő olyan *dolog*, amit elhelyezel egy grafikus felhasználói felületen (GUI-n). *Olyan dolog, amit a felhasználó lát, és műveleteket végez rajta.* A szövegmezők, gombok, gördíthető listák, választógombok stb. mind összetevők – valójában mindnyájan a `javax.swing.JComponent` osztályt bővítik.

A vezérlők valójában Swing-összetevők. Majdnem minden, amit egy grafikus felületre ragaszthatsz, a javax.swing.JComponent osztály leszármazottja.

Az összetevők egymásba ágyazhatók

A Swingben lényegében *minden* összetevő képes arra, hogy más összetevőket tároljon. Más szavakkal, *szinte bármit beletehetsz bármi másba.* A legtöbbször azonban olyan *interaktív felhasználói* összetevőket fogsz elhelyezni, mint a gombok és a listák, olyan *háttérösszetevőkben*, mint a keretek és a táblák (panelek). Bár *lehetséges* mondjuk egy táblát is elhelyezni egy gombban, azért az elég bizarr lenne, és nem nyernél vele díjat a programok használhatóságát mérő versenyeken.

Mindazonáltal a JFrame kivételével az *interaktív* összetevők és a *háttérösszetevők* közötti különbségtétel mesterséges. Egy JPanel például általában más összetevők csoportosításához szolgál háttérként, de egy JPanel is lehet interaktív: más összetevőkhöz hasonlóan a JPanel eseményeire is feliratkozhatsz, beleértve az egérkattintásokat és a billentyűleütéseket.

A grafikus felhasználói felületek létrehozásának négy lépése:

1. Létrehozol egy ablakot (JFrame-et):

```
JFrame frame = new JFrame();
```

2. Létrehozol egy összetevőt (gombot, szövegmezőt stb.):

```
JButton button = new JButton("kattints rám");
```

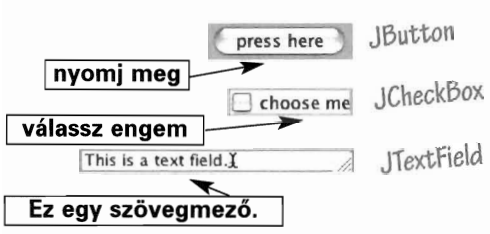
3. Hozzáadod az összetevőt a kerethez:

```
frame.getContentPane().add(BorderLayout.EAST, button);
```

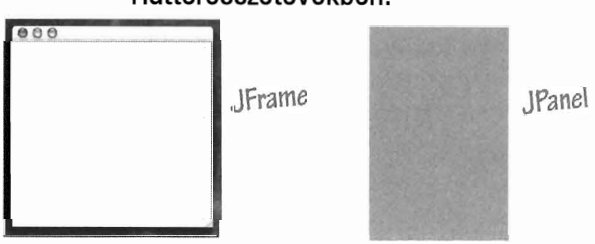
4. Megjeleníted a keretet (méretet adsz neki, és láthatóvá teszed):

```
frame.setSize(300,300);
frame.setVisible(true);
```

Interaktív összetevők elhelyezése...



Háttérösszetevőkben:



Elrendezéskezelők

Az elrendezéskezelő egy olyan Java-objektum, amely egy bizonyos összetevőhöz – majdnem mindig egy *háttérösszetevőhöz* – kapcsolódik. Az elrendezéskezelő szabályozza a hozzá társított összetevőn *belül* található összetevőket, vagyis ha egy keret egy panelt tartalmaz, a panel pedig egy gombot, a panel elrendezéskezelője határozza meg a gomb méretét és helyét, míg a keret elrendezéskezelője a panel hasonló jellemzőit. A gombnak ugyanakkor nincs szüksége elrendezéskezelőre, mert a gomb nem tárol más összetevőket.

Ha egy panel öt elemet tartalmaz, azok méretét és helyét kivétel nélkül a panel elrendezéskezelője szabja meg, akkor is, ha mind az öt elem rendelkezik saját elrendezéskezelővel. Ha viszont az öt elem *további* elemeket tartalmaz, *azok* elhelyezését az őket tároló elem elrendezéskezelője szabályozza.

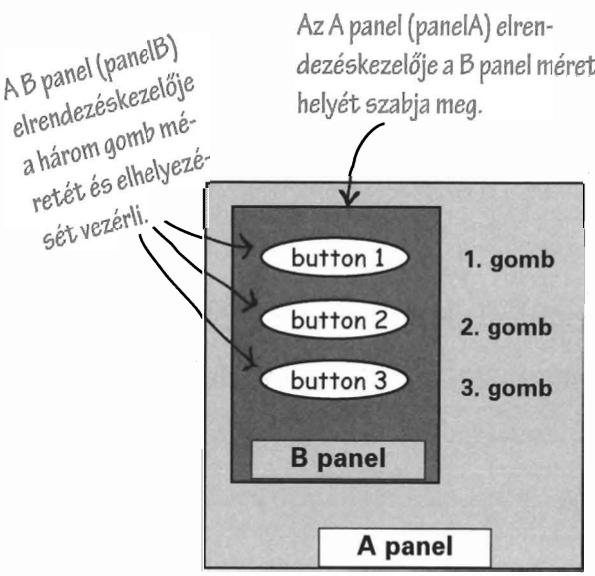
Amikor *tárolásról* beszélünk, valójában *hozzáadást* értünk alatta – egy panel úgy *tárolhat* egy gombot, hogy a gombot *hozzáadod* a panelhez, valahogy így:

```
myPanel.add(button);
```

Az elrendezéskezelőknek több fajtája létezik, és minden háttérösszetevőnek saját elrendezéskezelője lehet. Az elrendezéskezelők saját szabályokat („házirendet", policy) követnek az elrendezések (layout) felépítésekor: az egyik például ragaszkodhat hozzá, hogy egy panel minden összetevője azonos méretű és egy rácsban elrendezett legyen, míg a másik megengedheti az összetevőknek, hogy saját maguk határozzák meg a méretüket, de mindenképpen függőlegesen egymásra pakolja őket. Íme egy példa az elemek egymásba ágyazására:

```
JPanel panelA = new JPanel();
JPanel panelB = new JPanel();
panelB.add(new JButton("button 1"));
panelB.add(new JButton("button 2"));
panelB.add(new JButton("button 3"));
panelA.add(panelB);
```





Az A panel elrendezéskezelője SEMMIT sem mond a három gombról. A hierarchia csak egy szintű: az A panel elrendezéskezelője csak a közvetlenül az A panelhez adott elemeket vezérli, a panelbe beágyazott összetevők közül egyikkel sem foglalkozik.

Mi alapján dönt az elrendezéskezelő?

A különböző elrendezéskezelők más-más szabályokat követnek az összetevők elrendezésekor (például rácsba rendezhetik, azonos méretűre állíthatják, vagy egymás alá pakolhatják azokat, és így tovább), az elrendezett összetevőknek azonban mindig van *valamennyi* beleszólásuk a dologba. Egy háttérösszetevő elrendezése általában valahogy így zajlik:



Elrendezési forgatókönyv:

1. Létrehozol egy panelt, és hozzáadsz három gombot.
2. A panel elrendezéskezelője mindegyik gombtól megkérdezi, hogy mekkora szeretne lenni.
3. A panel elrendezéskezelője az elrendezési szabályai alapján eldönti, hogy részben vagy egészben figyelembe veszi-e a gombok kívánságait, vagy teljesen figyelmen kívül hagyja azokat.
4. A panelt hozzáadod egy kerethez.
5. A keret elrendezéskezelője megkérdezi a panelt, hogy mekkora szeretne lenni.
6. A keret elrendezéskezelője az elrendezési szabályai alapján eldönti, hogy részben vagy egészben figyelembe veszi-e a panel kívánságait, vagy teljesen figyelmen kívül hagyja azokat.

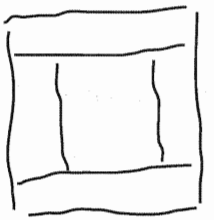
A különböző elrendezéskezelők más-más szabályokat követnek

Egyes elrendezéskezelők tiszteletben tartják, hogy egy összetevő mekkora szeretne lenni. Ha egy gomb mondjuk 30 x 50 képpont nagyságot szeretne, az elrendezéskezelő ekkora helyet foglal le a számára. Más elrendezéskezelők csak részben veszik figyelembe az összetevő által kívánt méretet, és ha a gomb 30 x 50 képpontos méretet akar, a méretét 30 x annyi képpontra állítják, amilyen széles a gomb hátterét képező *panel*. Megint más elrendezéskezelők csak az elrendezett összetevők *legnagyobbikának* kívánságait veszik figyelembe, míg a panel többi összetevőjét azonos méretűre állítják. Egyes esetekben az elrendezéskezelő munkája nagyon bonyolult lehet, a legtöbbször azonban ahhoz, hogy kitaláld, hogy az elrendezéskezelő valószínűleg mit fog tenni, elég, ha ismered a szabályait.

A Három Nagy elrendezéskezelő – keretek, folyásirány és dobozok

BorderLayout

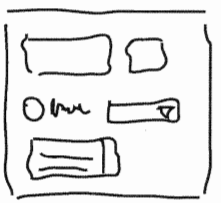
A BorderLayout (keretes elrendezés) kezelők a háttérösszetevőt öt területre (régióra) osztják. Egy BorderLayout kezelő által vezérelt háttérhez területenként csak egy összetevőt adhatsz hozzá, és a BorderLayout kezelő által elrendezett összetevők általában nem azt a méretet kapják, amit szeretnének. A **keretek (frame) alapértelmezett elrendezéskezelője a BorderLayout!**



területenként egy összetevő

FlowLayout

A FlowLayout (folyó elrendezés) kezelő úgy működik, mint egy szövegszerkesztő, csak éppen szavak helyett összetevőkkel dolgozik. Minden összetevő olyan méretű lesz, amilyen lenni szeretne, és az összetevők balról jobbra, a hozzáadásuk sorrendjében fognak elhelyezkedni, automatikus tördeléssel, ami azt jelenti, hogy ha egy összetevő nem fér el vízszintesen, akkor az elrendezés következő „sorába" kerül. A **FlowLayout a panelek alapértelmezett elrendezéskezelője.**



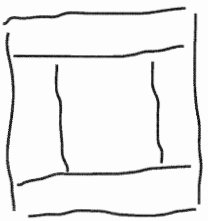
az összetevők balról jobbra rendeződnek, szükség esetén új sorba törve

BoxLayout

A BoxLayout (dobozelrendezés) kezelő annyiban hasonlít a FlowLayout kezelőre, hogy az összetevőket az általuk kívánt méretre állítja, és a hozzáadásuk sorrendjében helyezi el őket. A FlowLayout kezelővel ellentétben azonban a BoxLayout kezelő függőlegesen is el tudja helyezni az összetevőket (vízszintesen is, de minket általában a függőleges elrendezés érdekel). A dobozelrendezés olyan, mint a folyó elrendezés, de automatikus „összetevő-tördelés" helyett egyfajta „összetevő-elrendezési kulcsot" kell beszúrnod, ami arra **kényszeríti** az összetevőket, hogy új sort kezdjenek.



az összetevők felülről lefelé rendeződnek, és soronként egy összetevőt kapsz



A BorderLayout kezelő öt területet határoz meg: east (kelet), west (nyugat), north (észak), south (dél) és center (közép)

Adjunk egy gombot a keleti területhez:

```
import javax.swing.*;
import java.awt.*;
public class Button1 {

    public static void main (String[] args) {
        Button1 gui = new Button1();
        gui.go();
    }

    public void go() {
        JFrame frame = new JFrame();
        JButton button = new JButton("click me");
        frame.getContentPane().add(BorderLayout.EAST, button);
        frame.setSize(200,200);
        frame.setVisible(true);
    }
}
```

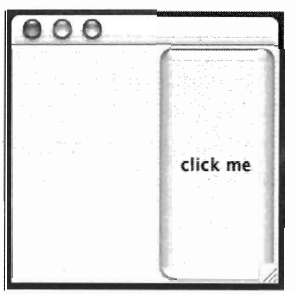
A BorderLayout a java.awt csomagban található

Itt határozod meg a területet



Agygyúró

Honnan vette a BorderLayout kezelő ezt a méretet a gomb számára?
Milyen tényezőket kell az elrendezéskezelőnek figyelembe vennie?
Miért nem szélesebb vagy magasabb a gomb?



Figyeld, mi történik, ha a gomb felirata hosszabb…

```
public void go() {
    JFrame frame = new JFrame();
    JButton button = new JButton("click like you mean it");
    frame.getContentPane().add(BorderLayout.EAST, button);
    frame.setSize(200,200);
    frame.setVisible(true);
}
```

Csak a gomb szövegét változtattuk meg





Button objektum



A gomb megkapja a kívánt szélességet, de a magasságot nem.



Button objektum

Próbáljunk ki egy gombot az ÉSZAKI területen:

```
public void go() {
    JFrame frame = new JFrame();
    JButton button = new JButton("There is no spoon...");
    frame.getContentPane().add(BorderLayout.NORTH, button);
    frame.setSize(200,200);
    frame.setVisible(true);
}
```



A gomb olyan magas, amilyen szeretett volna lenni, de a szélessége a keretével egyezik meg.

Most vegyük rá a gombot, hogy kérjen nagyobb magasságot

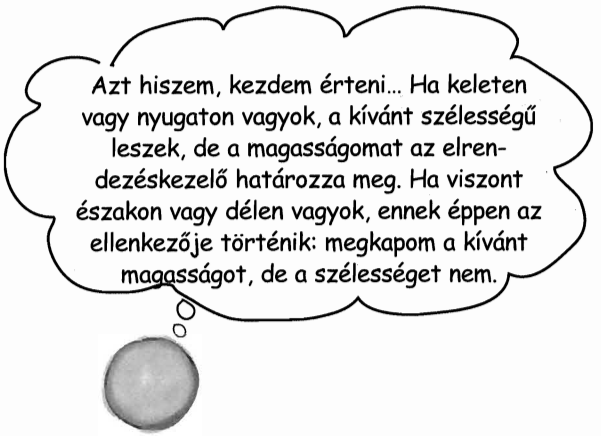
Ezt hogyan lehet elérni? A gomb már olyan széles, amilyen széles csak lehet – a keret szélességével megegyező. Viszont ha nagyobb betűket használsz rajta, a magasságát növelheted.

```
public void go() {
    JFrame frame = new JFrame();
    JButton button = new JButton("Click This!");
    Font bigFont = new Font("serif", Font.BOLD, 28);
    button.setFont(bigFont);
    frame.getContentPane().add(BorderLayout.NORTH, button);
    frame.setSize(200,200);
    frame.setVisible(true);
}
```

A nagyobb betűméret arra kényszeríti a keretet, hogy több helyet biztosítson a gomb magasságának.



A szélesség ugyanaz marad, de a gomb most magasabb. Az északi terület megnyúlt, hogy az új magasságot igénylő gomb elférjen.



De mi történik a középső területen?

A középső terület azt kapja, ami marad!

(kivéve egy különleges esetet, amivel később foglalkozunk)

```
public void go() {
    JFrame frame = new JFrame();

    JButton east = new JButton("East");
    JButton west = new JButton("West");
    JButton north = new JButton("North");
    JButton south = new JButton("South");
    JButton center = new JButton("Center");

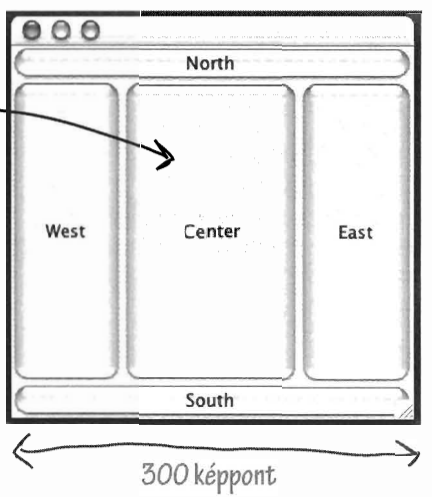
    frame.getContentPane().add(BorderLayout.EAST, east);
    frame.getContentPane().add(BorderLayout.WEST, west);
    frame.getContentPane().add(BorderLayout.NORTH, north);
    frame.getContentPane().add(BorderLayout.SOUTH, south);
    frame.getContentPane().add(BorderLayout.CENTER, center);

    frame.setSize(300,300);
    frame.setVisible(true);
}
```

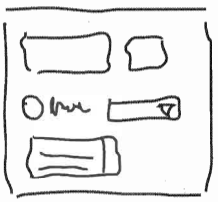
A középen levő összetevők annyi helyet kapnak, amennyi marad a keret kiterjedéséből (ami a példában 300 x 300 képpont)

A keleti és nyugati összetevők a kívánt szélességet kapják.

Az északi és déli összetevők a kívánt magasságot kapják.



Ha északon vagy délen helyezel el valamit, az a keret teljes szélességét elfoglalja, így a keleti és nyugati területek nem lesznek olyan magasak, mint amilyenek akkor lehetnének, ha az északi és déli területek üresek lennének.



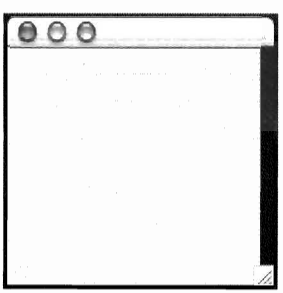
A FlowLayout kezelőt az összetevők folyása érdekli:

balról jobbra, felülről lefelé, a hozzáadás sorrendjében

Adjunk egy panelt a keleti területhez:

A JPanel-ek alapértelmezett elrendezéskezelője a FlowLayout. Amikor egy panelt egy kerethez adsz, a panel méretét és elhelyezését továbbra is a BorderLayout kezelő határozza meg, a *panelen belüli elemek* (más szavakkal, azok az összetevők, amelyeket a `panel.add(aComponent)` hívással hozzáadsz a panelhez) azonban a panel FlowLayout kezelőjének hatálya alá kerülnek. Először egy üres panelt adunk a keret keleti területéhez, majd a következő oldalakon különféle elemekkel egészítjük ki a panelt.

A panel nem tartalmaz semmit, így nem kell neki nagy szélesség a keleti területen.



```
import javax.swing.*;
import java.awt.*;

public class Panel1 {

    public static void main (String[] args) {
        Panel1 gui = new Panel1();
        gui.go();
    }

    public void go() {
        JFrame frame = new JFrame();
        JPanel panel = new JPanel();
        panel.setBackground(Color.darkGray);
        frame.getContentPane().add(BorderLayout.EAST, panel);
        frame.setSize(200,200);
        frame.setVisible(true);
    }
}
```

A panelt szürkére festjük, hogy lássuk, hol van a kereten belül.

Adjunk egy gombot a panelhez:

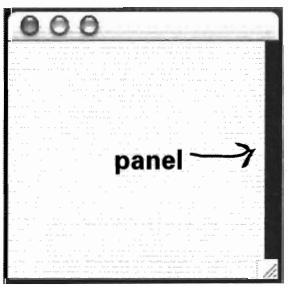
```
public void go() {
      JFrame frame = new JFrame();
      JPanel panel = new JPanel();
      panel.setBackground(Color.darkGray);

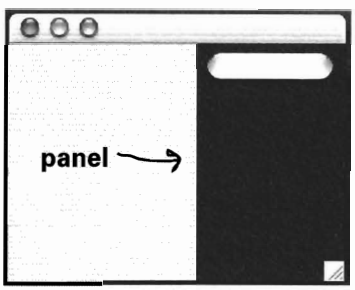
      JButton button = new JButton("shock me");

      panel.add(button);
      frame.getContentPane().add(BorderLayout.EAST, panel);

      frame.setSize(250,200);
      frame.setVisible(true);
}
```

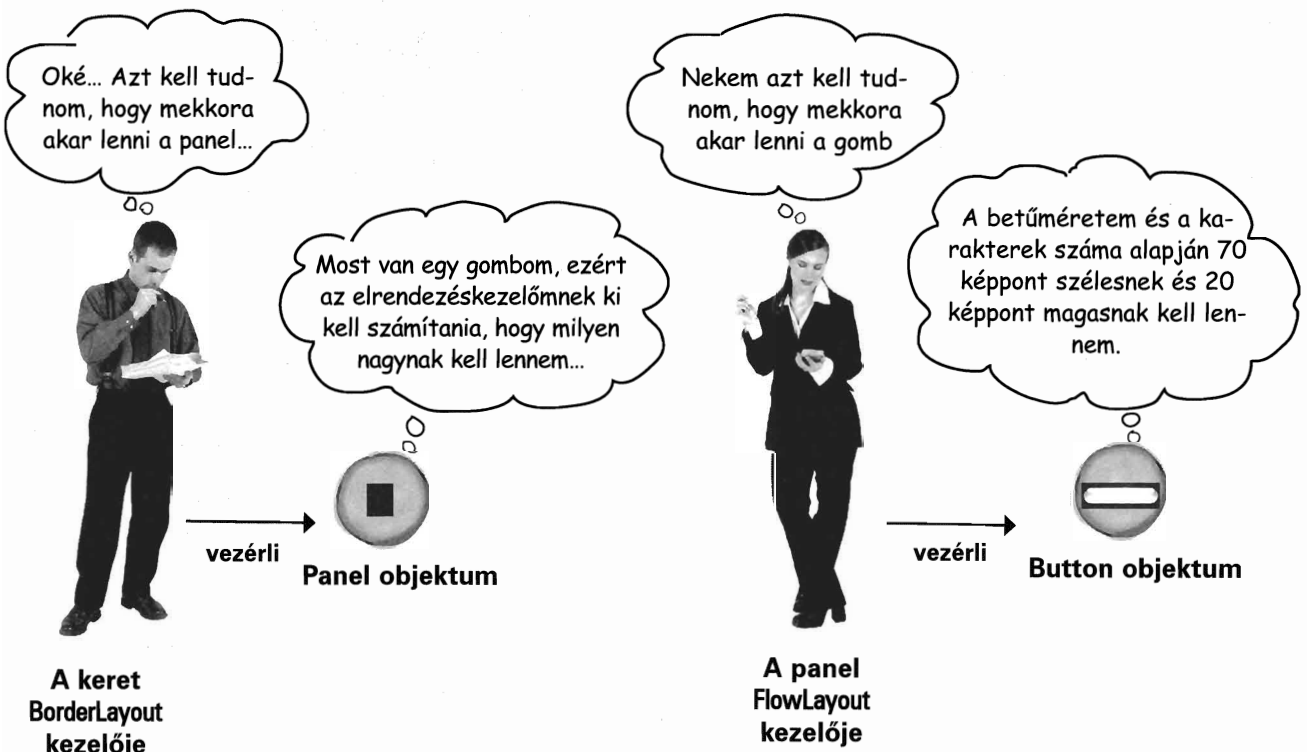
A gombot a panelhez adjuk, a panelt pedig a kerethez. A gombot a panel elrendezéskezelője (FlowLayout), a panelt pedig a kereté (BorderLayout) vezérli.





A panel megnő!

A gomb mindkét irányban a kívánt méretet veszi fel, mivel a panel folyó elrendezést használ, a gomb pedig a panel (és nem a keret) része.



Mi történik, ha KÉT gombot adunk a panelhez?

```
public void go() {
    JFrame frame = new JFrame();
    JPanel panel = new JPanel();
    panel.setBackground(Color.darkGray);

    JButton button = new JButton("shock me");
    JButton buttonTwo = new JButton("bliss");

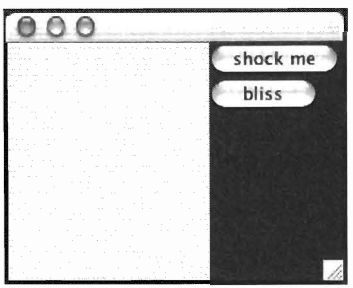
    panel.add(button);
    panel.add(buttonTwo);

    frame.getContentPane().add(BorderLayout.EAST, panel);
    frame.setSize(250,200);
    frame.setVisible(true);
}
```

KÉT gombot hozunk létre

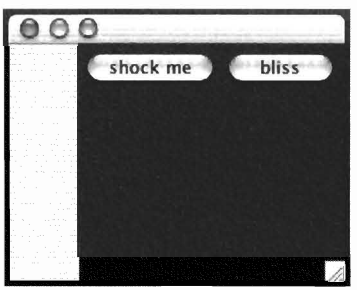
MINDKÉT gombot a panelhez adjuk

amit akartunk:



Azt akartuk, hogy a gombok egymás fölé kerüljenek.

amit kaptunk:



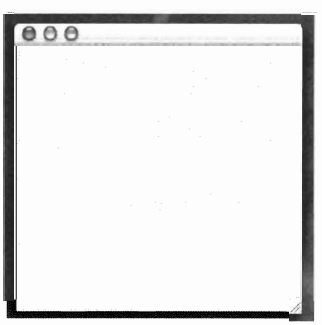
A panel megnőtt, hogy a két gomb elférjen egymás mellett.

Vedd észre, hogy a „bliss" gomb kisebb a „shock me" gombnál... így működik a folyó elrendezés. A gomb pontosan annyit kap, amennyire szüksége van (és nem többet).

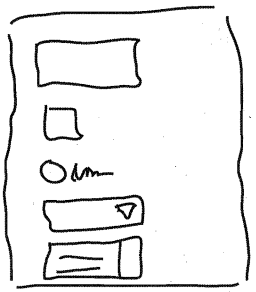
Hegyezd ki a ceruzád!

Ha a fenti kódot az alábbira módosítanánk, hogyan festene a grafikus felület?

```
JButton button = new JButton("shock me");
JButton buttonTwo = new JButton("bliss");
JButton buttonThree = new JButton("huh?");
panel.add(button);
panel.add(buttonTwo);
panel.add(buttonThree);
```



Rajzold ide, hogy szerinted hogy nézne ki a grafikus felület a bal oldali kód futtatásakor. (Aztán próbáld ki!)



A dobozelrendezés megment, mert az összetevőket akkor is egymás fölé helyezi, ha egymás mellett is elférnének.

A folyó elrendezéstől (FlowLayout) eltérően a dobozelrendezés (BoxLayout) akkor is képes új sorba kényszeríteni az összetevőket, ha vízszintesen is elég hely van a számukra.

Ehhez azonban a panel elrendezéskezelőjét az alapértelmezett FlowLayout-ról BoxLayout-ra kell módosítanod:

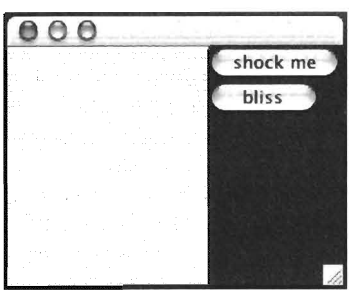
```
public void go() {
    JFrame frame = new JFrame();
    JPanel panel = new JPanel();
    panel.setBackground(Color.darkGray);

    panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));

    JButton button = new JButton("shock me");
    JButton buttonTwo = new JButton("bliss");
    panel.add(button);
    panel.add(buttonTwo);
    frame.getContentPane().add(BorderLayout.EAST, panel);
    frame.setSize(250,200);
    frame.setVisible(true);
}
```

Az elrendezéskezelőt egy új BoxLayout-példányra cseréljük.

A BoxLayout konstruktornak tudnia kell, milyen összetevőt kell elrendeznie (a panelt), és melyik tengelyen (a függőleges elrendezéshez az Y_AXIS-t használjuk).



Figyeld meg, hogy a panel ismét keskenyebb, mert nem kell mindkét gombnak vízszintesen elférnie rajta. A panel tehát azt közli a kerettel, hogy csak a nagyobbik gomb („shock me") elhelyezéséhez elegendő helyet igényel.

Nincsenek

hülye kérdések

K: Miért nem lehet közvetlenül elemeket adni egy kerethez úgy, mint egy panelhez?

V: A JFrame azért különleges, mert ő jelenti a kapcsolatot ahhoz, hogy valami megjelenjen a képernyőn. Minden Swing-összetevő tisztán Java alapú, a JFrame-nek azonban kapcsolódnia kell a háttérben működő operációs rendszerhez, hogy elérje a megjelenítőt. A tartalomtáblára gondolj úgy, mint egy 100%-ig tiszta Java-rétegre, ami a JFrame *tetején* ül. Vagy tekintsd úgy, mintha a JFrame egy ablakkeret lenne, a tartalomtábla pedig az ablaküveg. A tartalomtáblát ki is cserélheted egy saját JPanel-re, hogy a te JPanel-ed legyen a keret tartalomtáblája:

```
myFrame.setContentPane(myPanel);
```

K: Megváltoztathatom a keret elrendezéskezelőjét? Mit tehetek, ha azt szeretném, hogy a keret folyó elrendezést használjon keretes helyett?

V: Ezt a legegyszerűbben úgy érheted el, hogy létrehozol egy panelt, felépíted a GUI-t a panelben úgy, ahogy szeretnéd, majd ezt a panelt teszed a keret tartalomtáblájává az előző válaszban látott kód segítségével (ahelyett, hogy az alapértelmezett tartalomtáblát használnád).

K: És ha más méretet szeretnék beállítani? Létezik setSize() – beállítMéret – tagfüggvény az összetevőkhöz?

V: Igen, létezik setSize() tagfüggvény, de az elrendezéskezelők figyelmen kívül hagyják. Az összetevő *saját maga által kívánt mérete* – „előnyben részesített méret" (preferred size) – és az *általad* kívánt méret nem ugyanaz. Az előnyben részesített méret az összetevő ténylegesen *igényelt* méretén alapul (ezt maga az összetevő határozza meg). Az elrendezéskezelő meghívja az összetevő getPreferredSize() tagfüggvényét, *ezt* a tagfüggvényt pedig *nem érdekli*, hogy korábban meghívtad-e a setSize() tagfüggvényt az összetevőre.

K: Nem helyezhetem az elemeket egyszerűen oda, ahová akarom? Ki lehet kapcsolni az elrendezéskezelőket?

V: Aha. Ha összetevőnként szeretnéd elrendezni a felületet, meghívhatod a setLayout(null)-t, majd bedrótozhatod a képernyőn elfoglalt pontos helyet és méretet. Hosszú távon azonban majdnem mindig könnyebb elrendezéskezelőket használni.

Újratöltve

- Az elrendezéskezelők a más összetevőkbe beágyazott összetevők méretét és elhelyezését szabályozzák.
- Amikor egy összetevőt egy másik összetevőhöz adsz (az utóbbit néha *háttérösszetevőnek* hívják, de ez nem hivatalos elnevezés), a hozzáadott összetevőt a *háttérösszetevő* elrendezéskezelője fogja vezérelni.
- Az elrendezéskezelők megkérdezik az összetevőket, hogy milyen méretet részesítenek előnyben, mielőtt döntést hoznának az elrendezésről. Az elrendezéskezelő a szabályaitól függően részben vagy egészben tiszteletben tarthatja az összetevő kívánságait, de teljesen figyelmen kívül is hagyhatja azokat.
- A BorderLayout (keretes elrendezés) kezelő használatakor öt terület egyikéhez adhatod hozzá az összetevőket. Az összetevő hozzáadásakor meg kell adnod a területet, az alábbi utasításformát követve:

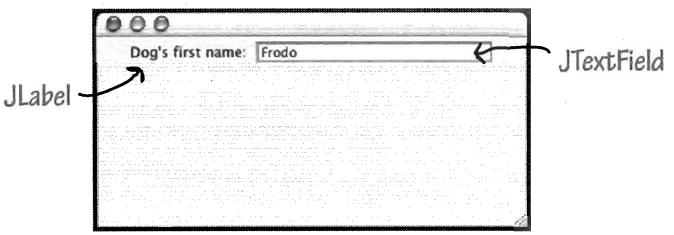
```
add(BorderLayout.EAST, panel);
```

- A keretes elrendezésben az északi és déli területen levő összetevők a kívánt magasságúak lesznek, de nem olyan szélességűek. A keleti és nyugati terület összetevői ezzel szemben a kívánt szélességűek lesznek, de nem olyan magasak. A középen levő összetevő a maradék hellyel gazdálkodhat (hacsak nem használod a **pack()**-ot).
- A pack() tagfüggvény olyan, mint egy zsugorfólia az összetevők számára: a középső összetevőnek megadja a teljes előnyben részesített méretét, majd meghatározza a keret méretét a középső területből kiindulva, és a többi terület tartalmának megfelelően felépíti a maradékot.
- A FlowLayout (folyó elrendezés) az összetevőket balról jobbra, felülről lefelé rendezi el, a hozzáadásuk sorrendjében, az összetevőket csak akkor törve új sorba, ha az összetevő vízszintesen nem fér el.
- A folyó elrendezés mindkét irányban megadja az összetevőknek az általuk előnyben részesített méretet.
- A BoxLayout (dobozelrendezés) lehetővé teszi, hogy az összetevőket függőlegesen egymásra pakolva helyezd el, akkor is, ha egyébként elférnének egymás mellett. A folyó elrendezéshez hasonlóan a dobozelrendezés is mindkét irányban megadja az összetevőknek az általuk előnyben részesített méretet.
- A keretek alapértelmezett elrendezéskezelője a BorderLayout, a paneleké pedig a FlowLayout.
- Ha azt szeretnéd, hogy egy panel folyó elrendezés helyett mást használjon, meg kell hívnod a **setLayout()** tagfüggvényt a panelre.

Játék a Swing-összetevőkkel

Most, hogy elsajátítottad az elrendezéskezelők használatának alapjait, kipróbálhatunk néhányat a leggyakrabban használt összetevők közül: egy egyszerű szövegmezőt, egy gördíthető többsoros szövegmezőt, egy jelölőnégyzetet és egy listamezőt. Természetesen nem mutatjuk meg az egyes összetevők teljes alkalmazásprogramozási felületét, csak a legfontosabb tudnivalókat, hogy megtehesd az első lépéseket.

JTextField



Konstruktorok

```
JTextField field = new JTextField(20);
```

A „20" 20 oszlopot, nem 20 képpontot jelent. Ez határozza meg a szövegmező előnyben részesített méretét.

```
JTextField field = new JTextField("Your name");
```

Hogyan használhatod?

1. Szöveg kiolvasása a mezőből

```
System.out.println(field.getText());
```

2. Szöveg beírása a mezőbe

```
field.setText("whatever");
field.setText("");
```

Ez törli a mező tartalmát.

3. Műveleti esemény kiolvasása, amikor a felhasználó lenyomja az ENTER billentyűt

```
field.addActionListener(myActionListener);
```

Fel is iratkozhatsz a billentyűeseményekre, ha tényleg értesítést szeretnél minden billentyűlenyomásról.

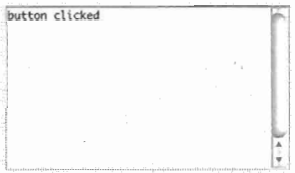
4. A szöveg kijelölése a mezőben

```
field.selectAll();
```

5. A kurzor visszahelyezése a mezőbe (hogy a felhasználó gépelni kezdhessen)

```
field.requestFocus();
```

JTextArea



A JTextField-től eltérően a JTextArea több sornyi szöveget is tartalmazhat. Egy többsoros szövegmező létrehozása pár beállítást igényel, mivel alaphelyzetben nem járnak hozzá gördítősávok, illetve automatikus sortörés. Ahhoz, hogy egy JTextArea gördíthető legyen, egy JScrollPane (gördülőtábla) összetevőben kell elhelyezned. A JScrollPane olyan objektum, amely nagyon szeret gördülni, és örömmel gondoskodik a szövegmező görgetéséről.

Konstruktor

A „10" 10 sort jelent (ez lesz az előnyben részesített magasság)

A „20" 20 oszlopot jelent (ez lesz az előnyben részesített szélesség)

```
JTextArea text = new JTextArea(10,20);
```

Hogyan használhatod?

1. Csak függőleges gördítősáv hozzáadása a mezőhöz

Létrehozunk egy JScrollPane-t, és átadjuk neki a többsoros szövegmezőt, amelyet görgetni fog.

Bekapcsoljuk az automatikus tördelést.

Arra utasítjuk a gördülőtáblát, hogy csak függőleges gördítősávot alkalmazzon.

```
JScrollPane scroller = new JScrollPane(text);
text.setLineWrap(true);

scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

panel.add(scroller);
```

Fontos!!! A szövegmezőt a gördülőtáblához adjuk hozzá (a gördülőtábla konstruktorán keresztül), majd a gördülőtáblát a panelhez. A szövegmezőt nem közvetlenül adod a panelhez!

2. A mezőben levő szöveg cseréje

```
text.setText("Not all who are lost are wandering");
```

3. A mezőben levő szöveg kiegészítése

```
text.append("button clicked");
```

4. A szöveg kijelölése a mezőben

```
text.selectAll();
```

5. A kurzor visszahelyezése a mezőbe (hogy a felhasználó gépelni kezdhessen)

```
text.requestFocus();
```

Példa a JTextArea használatára

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;


public class TextArea1 implements ActionListener {

    JTextArea text;

    public static void main (String[] args) {
       TextArea1 gui = new TextArea1();
       gui.go();
   }


   public void go() {
      JFrame frame = new JFrame();
      JPanel panel = new JPanel();
      JButton button = new JButton("Just Click It");
      button.addActionListener(this);
      text = new JTextArea(10,20);
      text.setLineWrap(true);

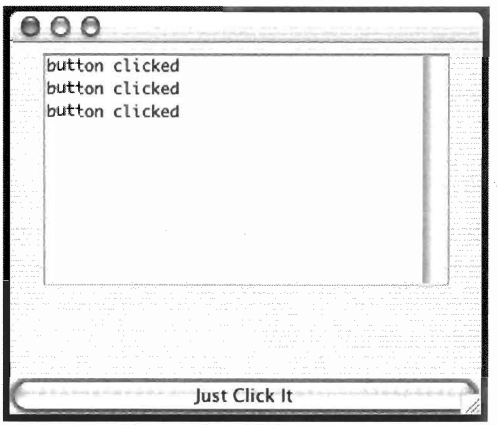
      JScrollPane scroller = new JScrollPane(text);
      scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
      scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

      panel.add(scroller);

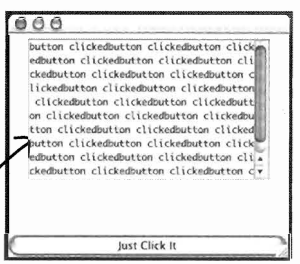
      frame.getContentPane().add(BorderLayout.CENTER, panel);
      frame.getContentPane().add(BorderLayout.SOUTH, button);

      frame.setSize(350,300);
      frame.setVisible(true);
   }

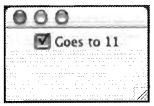
   public void actionPerformed(ActionEvent ev) {
      text.append("button clicked \n ");
   }
}
```



Beszúrunk egy új sort, hogy a szavak minden gombnyomásra külön sorba kerüljenek, különben összefutnának.



JCheckBox



Konstruktor

```
JCheckBox check = new JCheckBox("Goes to 11");
```

Hogyan használhatod?

1. Elem eseményeinek (be- és kikapcsolás) figyelése

```
check.addItemListener(this);
```

2. Az esemény kezelése (és annak megállapítása, hogy az elem be van-e kapcsolva vagy sem)

```
public void itemStateChanged(ItemEvent ev) {
   String onOrOff = "off";
   if (check.isSelected()) onOrOff = "on";
   System.out.println("Check box is " + onOrOff);
}
```

3. Be- vagy kikapcsolás kódból

```
check.setSelected(true);
check.setSelected(false);
```

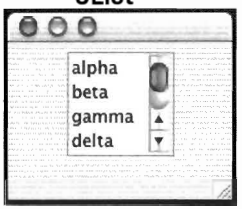
Nincsenek

hülye kérdések

K: Nekem úgy tűnik, hogy az elrendezéskezelők több gondot okoznak, mint amennyit nyerünk velük. Ha ilyen körülményes a használatuk, nem lenne egyszerűbb, ha minden összetevőnek bedrótoznám a méretét és a koordinátáit?

V: Egy elrendezéskezelőből valóban nem mindig egyszerű pontosan a kívánt elrendezést kicsikarni, de gondold végig, mi mindent végez el helyetted. Még egy olyan látszólag könnyű feladat, mint az elemek helyének meghatározása a képernyőn, is bonyolult lehet.
Az elrendezéskezelő például gondoskodik róla, hogy az összetevőid ne takarják el egymást, vagyis tudja, hogyan kezelje a közöttük levő (illetve az összetevők és a keret széle közötti) helyet. Ezt persze te is megteheted, de mi van, ha azt szeretnéd, hogy az összetevők szorosan egymáshoz simuljanak? Saját kezűleg a megfelelő helyre teheted ugyan őket, de az elrendezés csak a te JVM-eden lesz pontos.
Miért? Mert az összetevők rendszerről rendszerre különböznek kissé, különösen ha a háttérben működő rendszer beépített stílusát használják. Elég, ha az olyan finomságok, mint a gombok pereme, mások egy kicsit, és az egyik rendszeren katonásan egymás mellett sorakozó összetevők a másik rendszeren hirtelen egymásra csúsznak.
És akkor még nem ejtettünk szót az elrendezéskezelők igazi Nagy Előnyéről. Gondold végig, mi történik, ha a felhasználó átméretezi az ablakot, vagy ha a grafikus felület összetevői dinamikusan változnak! Ha neked kellene állandóan újrarendezned az összetevőket, amikor csak megváltozik a háttérösszetevő mérete vagy tartalma… jobb, ha bele sem gondolsz.

JList



A JList konstruktora bármilyen típusú objektumtömböt elfogad – nem kell String-eknek lenniük, de a listában a karakterlánc-megfelelőjük fog megjelenni.

Konstruktor

```
String [] listEntries = {"alpha", "beta", "gamma", "delta",
                         "epsilon", "zeta", "eta", "theta "};

list = new JList(listEntries);
```

Hogyan használhatod?

Ezt pontosan ugyanúgy csináljuk, mint a JTextArea esetében: létrehozunk egy JScrollPane-t (és beletesszük a listamezőt), majd a gördülőtáblát (NEM a listamezőt) hozzáadjuk a panelhez.

1. Függőleges gördítősáv hozzáadása

```
JScrollPane scroller = new JScrollPane(list);
scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

panel.add(scroller);
```

2. A gördítés nélkül megjelenő sorok számának beállítása

```
list.setVisibleRowCount(4);
```

3. A felhasználó által egyszerre kijelölhető elemek számának EGYRE korlátozása

```
list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

4. Feliratkozás a listaelem-kijelölési eseményekre

```
list.addListSelectionListener(this);
```

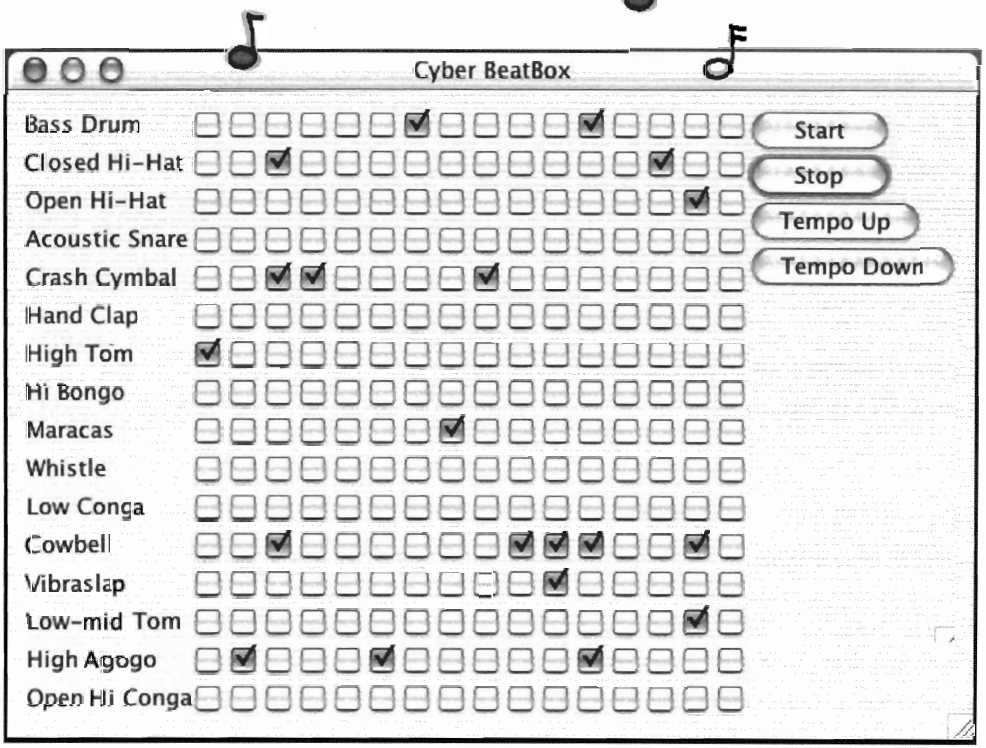
5. Az események kezelése (annak kiderítése, hogy a lista melyik elemét jelölték ki)

```
public void valueChanged(ListSelectionEvent lse) {
      if( !lse.getValueIsAdjusting()) {
        String selection = (String) list.getSelectedValue();
        System.out.println(selection);
      }
  }
```

Ha ezt az if vizsgálatot nem végzed el, az eseményt KÉTSZER kapod meg.

A getSelectedValue() valójában egy Object-et ad vissza – a listamezőkben nem csak String objektumok lehetnek.

Kódkonyha



Ez a rész nem kötelező. Elkészítjük a teljes BeatBox alkalmazást, grafikus felülettel, meg minden. Az objektumok mentéséről szóló következő fejezetben aztán megtanulod, hogyan mentheted és állíthatod vissza a dobmintákat, a hálózatkezelésről és a kapcsolatteremtésről szóló fejezetben pedig a BeatBox-ot működőképes csevegőügyféllé alakítod.

A BeatBox felépítése

A következő oldalakon a BeatBox második változatának teljes kódját megtalálod, az elindításra, leállításra és a tempó megváltoztatására szolgáló gombokkal együtt. A kód teljes, és hiánytalanul el van látva megjegyzésekkel, de álljon itt egy áttekintés:

1. Felépítünk egy grafikus felületet, amelyen 256 jelölőnégyzet (JCheckBox) található, amelyek kezdetben ki vannak kapcsolva; 16 szövegcímke (JLabel) a hangszerek neveihez; valamint 4 gomb.

2. Bejegyeztetünk egy ActionListener-t mind a négy gombhoz. Az egyes jelölőnégyzetekhez nincs szükség figyelőkre, mivel a minta hangját nem dinamikusan próbáljuk változtatni (vagyis nem azonnal, ahogy a felhasználó bekapcsol egy jelölőnégyzetet), hanem várunk, amíg a felhasználó a „start" gombra kattint, majd végigmegyünk mind a 256 jelölőnégyzeten, hogy kiolvassuk az állapotukat, és felépítsük a MIDI-számot.

3. Beállítjuk a MIDI-rendszert (ezt korábban már elvégeztük), beleértve egy sorozat-feldolgozó (Sequencer), egy utasítássorozat (Sequence) és egy sáv (track) létrehozását. A Java 5.0-ban megjelent új sorozat-feldolgozó tagfüggvényt, a setLoopCount()-ot fogjuk használni. Ez a tagfüggvény lehetővé teszi, hogy megadd, hogy hányszor szeretnéd ismételni az utasítássorozatot. Ezenkívül az utasítássorozat tempo tényezőjének segítségével gyorsítunk vagy lassítunk a tempón, és megőrizzük azt a ciklus ismétlései között.

4. Az igazi móka akkor indul, amikor a felhasználó a „start" gombra kattint. A „start" gomb eseménykezelő tagfüggvénye meghívja a buildTrackAndStart() tagfüggvényt, amelyben végigmegyünk mind a 256 jelölőnégyzeten (soronként, tehát egy-egy hangszer „kottájának" mind a 16 ütemén), hogy kiolvassuk az állapotukat, majd ezekből az információkból felépítünk egy MIDI-sávot (az előző fejezetben használt hasznos makeEvent() tagfüggvény segítségével). Ha a sáv elkészült, elindítjuk a sorozat-feldolgozót, ami addig folytatja a lejátszást (mivel ciklusban futtatjuk), amíg a felhasználó a „stop" gombra nem kattint.

```
import java.awt.*;
import javax.swing.*;
import javax.sound.midi.*;
import java.util.*;
import java.awt.event.*;


public class BeatBox {


    JPanel mainPanel;
    ArrayList<JCheckBox> checkboxList;
    Sequencer sequencer;
    Sequence sequence;
    Track track;
    JFrame theFrame;

    String[] instrumentNames = {"Bass Drum", "Closed Hi-Hat",
       "Open Hi-Hat","Acoustic Snare", "Crash Cymbal", "Hand Clap",
       "High Tom", "Hi Bongo", "Maracas", "Whistle", "Low Conga",
       "Cowbell", "Vibraslap", "Low-mid Tom", "High Agogo",
       "Open Hi Conga"};
    int[] instruments = {35,42,46,38,49,39,50,60,70,72,64,56,58,47,67,63};


    public static void main (String[] args) {
        new BeatBox2().buildGUI();
    }

    public void buildGUI() {
        theFrame = new JFrame("Cyber BeatBox");
        theFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        BorderLayout layout = new BorderLayout();
        JPanel background = new JPanel(layout);
        background.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));

        checkboxList = new ArrayList<JCheckBox>();
        Box buttonBox = new Box(BoxLayout.Y_AXIS);

        JButton start = new JButton("Start");
        start.addActionListener(new MyStartListener());
        buttonBox.add(start);

        JButton stop = new JButton("Stop");
        stop.addActionListener(new MyStopListener());
        buttonBox.add(stop);

        JButton upTempo = new JButton("Tempo Up");
        upTempo.addActionListener(new MyUpTempoListener());
        buttonBox.add(upTempo);

        JButton downTempo = new JButton("Tempo Down");
```

A jelölőnégyzeteket egy tömblistában tároljuk.

Ezek a hangszerek nevei String-tömbként, a grafikus felületen (egy-egy sorban) megjelenő szövegcímkék számára.

Ezek jelképezik a dob tényleges „billentyűit". A dobcsatorna olyan, mint egy zongora, csak éppen a „billentyűk" más-más hangú doboknak felelnek meg. A 35-ös szám például a basszusdobé, a 42-es a zárt állású lábciné, és így tovább.

Az „üres szegély" (empty border) margót biztosít a panel széle és az összetevők között – pusztán a vonzóbb látvány kedvéért.

Itt nincs semmi különleges, csak egy csomó GUI-kód. A nagy részét már láttad.

```
    downTempo.addActionListener(new MyDownTempoListener());
    buttonBox.add(downTempo);

    Box nameBox = new Box(BoxLayout.Y_AXIS);
    for (int i = 0; i < 16; i++) {
       nameBox.add(new Label(instrumentNames[i]));
    }

    background.add(BorderLayout.EAST, buttonBox);
    background.add(BorderLayout.WEST, nameBox);

    theFrame.getContentPane().add(background);

    GridLayout grid = new GridLayout(16,16);
    grid.setVgap(1);
    grid.setHgap(2);
    mainPanel = new JPanel(grid);
    background.add(BorderLayout.CENTER, mainPanel);

    for (int i = 0; i < 256; i++) {
        JCheckBox c = new JCheckBox();
        c.setSelected(false);
        checkboxList.add(c);
        mainPanel.add(c);
    } // ciklus vége

    setUpMidi();

    theFrame.setBounds(50,50,300,300);
    theFrame.pack();
    theFrame.setVisible(true);
} // tagfüggvény lezárása

public void setUpMidi() {
  try {
    sequencer = MidiSystem.getSequencer();
    sequencer.open();
    sequence = new Sequence(Sequence.PPQ,4);
    track = sequence.createTrack();
    sequencer.setTempoInBPM(120);

  } catch(Exception e) {e.printStackTrace();}
} // tagfüggvény lezárása
```

További GUI-beállító kód. Semmi említésre méltó.

Létrehozzuk a jelölőnégyzeteket, „false" értékre állítjuk őket (hogy ne legyenek bekapcsolva), és hozzáadjuk őket a tömblistához ÉS a GUI-panelhez.

A szokásos MIDI-beállítás a sorozat-feldolgozó, az utasítássorozat és a sáv számára. Semmi különös.

Itt történik a lényeg! A jelölőnégyzetek állapotát MIDI-eseményekké alakítjuk, és hozzáadjuk őket a sávhoz.

```
public void buildTrackAndStart() {
  int[] trackList = null;

  sequence.deleteTrack(track);
  track = sequence.createTrack();

    for (int i = 0; i < 16; i++) {
      trackList = new int[16];

      int key = instruments[i];

      for (int j = 0; j < 16; j++ ) {

          JCheckBox jc = (JCheckBox) checkboxList.get(j + (16*i));
          if ( jc.isSelected()) {
             trackList[j] = key;
          } else {
             trackList[j] = 0;
          }
       } // belső ciklus lezárása

       makeTracks(trackList);
       track.add(makeEvent(176,1,127,0,16));
    } // külső ciklus lezárása

    track.add(makeEvent(192,9,1,0,15));
    try {

       sequencer.setSequence(sequence);
        sequencer.setLoopCount(sequencer.LOOP_CONTINUOUSLY);
        sequencer.start();
        sequencer.setTempoInBPM(120);
    } catch(Exception e) {e.printStackTrace();}
} // a buildTrackAndStart tagfüggvény lezárása

public class MyStartListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        buildTrackAndStart();
    }
} // belső osztály lezárása
```

Létrehozunk egy 16 elemű tömböt, amelyben egy hangszer értékeit tároljuk a 16 ütemre. Ha egy hangszernek meg kell szólalnia egy adott ütemben, a tömb adott elemének értéke lesz a kulcs. Ha a hangszernek NEM kell megszólalnia az említett ütemben, nullát szúrunk be.

Megszabadulunk a régi sávtól, és létrehozunk egy újat.

Ezt mind a 16 SORRA (Bass, Congo stb.) elvégezzük.

Beállítjuk a „kulcsot", ami a hangszert jelképezi (Bass, Hi-Hat stb.) Az egyes hangszerek MIDI-számazonosítóit az instruments – hangszerek – tömb tárolja.

Ezt a sor minden ÜTEMÉRE elvégezzük.

Be van kapcsolva a jelölőnégyzet erre az ütemre? Ha igen, a tömb megfelelő rekeszébe (az adott ütemet jelképező rekeszbe) beszúrjuk a kulcsértéket, ha viszont nem, a hangszernek NEM kell megszólalnia ebben az ütemben, ezért nullát szúrunk be.

Ehhez a hangszerhez és mind a 16 ütemhez eseményeket hozunk létre, és hozzáadjuk azokat a sávhoz.

Mindig gondoskodni kell róla, hogy LEGYEN esemény a 16. ütemhez (a számozás 0-tól 15-ig tart), különben a BeatBox esetleg nem megy végig mind a 16 ütemen, mielőtt újrakezdené a ciklust.

Itt adhatod meg a ciklus ismétléseinek számát – ebben az esetben folyamatos ismétlést állítunk be.

ÉS MOST A LEJÁTSZÁS!

Az első a belső osztályok közül – figyelők a gombok számára. Nincs benne semmi szokatlan.

```
public class MyStopListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        sequencer.stop();
    }
} // belső osztály lezárása

public class MyUpTempoListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
       float tempoFactor = sequencer.getTempoFactor();
        sequencer.setTempoFactor((float)(tempoFactor * 1.03));
    }
} // belső osztály lezárása

public class MyDownTempoListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
       float tempoFactor = sequencer.getTempoFactor();
        sequencer.setTempoFactor((float)(tempoFactor * .97));
    }
} // belső osztály lezárása
```

A gombok többi belső figyelőosztálya.

A TempoFactor szabályozza a sorozat-feldolgozó tempóját, a megadott tényezővel felszorozva. Az alapértelmezés 1.0, tehát ütésenként +/- 3%-kal igazítunk a tempón.

Ez egyszerre egy hangszer számára hoz létre eseményeket, mind a 16 ütemre. Vagyis lehet egy int[] a basszusdobhoz, és a tömb egyes rekeszeiben vagy a hangszer kulcsa, vagy nulla áll majd. Ha nulla, a hangszernek nem kell megszólalnia az adott ütemben, más esetben viszont létrehozunk egy eseményt, és hozzáadjuk a sávhoz.

```
public void makeTracks(int[] list) {

   for (int i = 0; i < 16; i++) {
      int key = list[i];

      if (key != 0) {
         track.add(makeEvent(144,9,key, 100, i));
         track.add(makeEvent(128,9,key, 100, i+1));
      }
   }
}
```

Létrehozzuk a NOTE ON és NOTE OFF eseményeket, és hozzáadjuk azokat a sávhoz.

```
public  MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
    MidiEvent event = null;
    try {
        ShortMessage a = new ShortMessage();
        a.setMessage(comd, chan, one, two);
        event = new MidiEvent(a, tick);

    } catch(Exception e) {e.printStackTrace(); }
    return event;
}

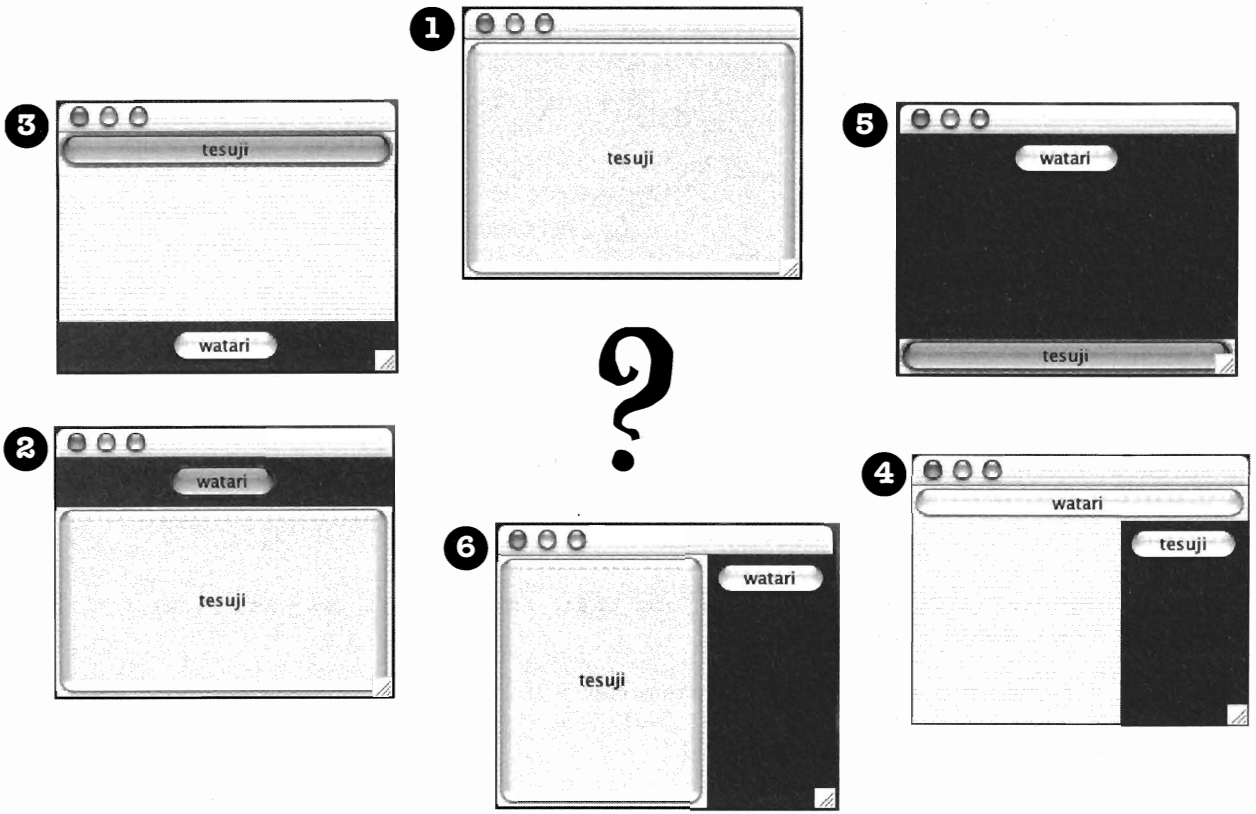
} // osztály lezárása
```

Ezt a segédfüggvényt már láttad az előző fejezet Kódkonyhájában. Nincs benne semmi újdonság.



Melyik kód melyik elrendezéshez tartozik?

Az alábbi hat képernyőből öt egy-egy kódtöredék eredménye, amelyeket a következő oldalon láthatsz. Párosítsd össze az öt kódtöredéket a megfelelő elrendezéssel!



Kódtöredékek

D

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.NORTH,panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

B

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
frame.getContentPane().add(BorderLayout.EAST, panel);
```

C

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

A

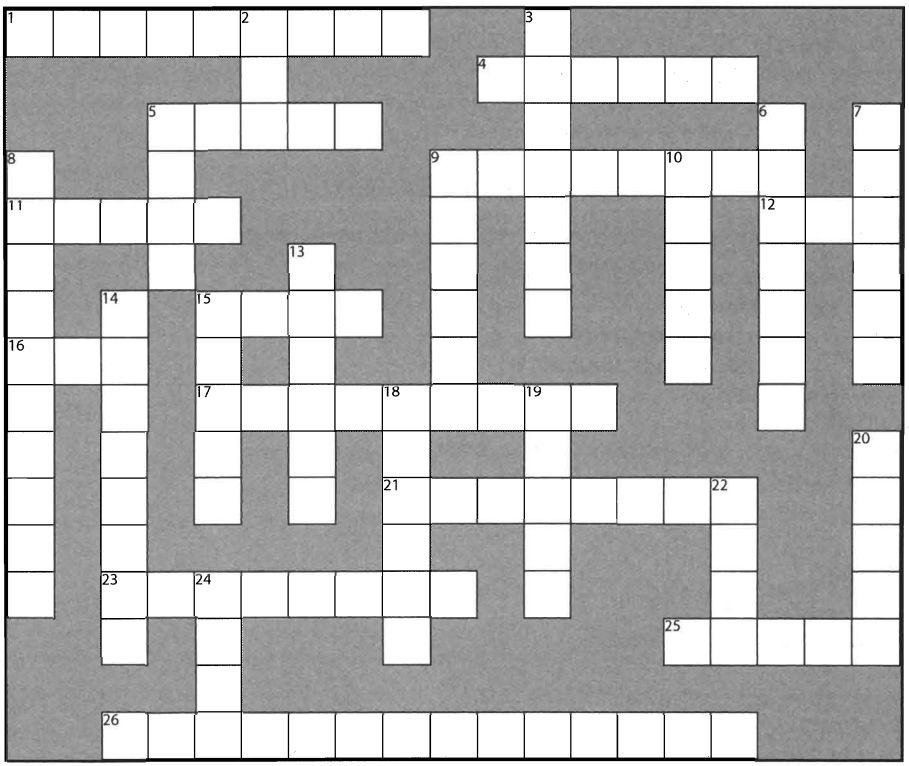
```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(button);
frame.getContentPane().add(BorderLayout.NORTH,buttonTwo);
frame.getContentPane().add(BorderLayout.EAST, panel);
```

E

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.SOUTH,panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.NORTH,button);
```



GUI-keresztrejtvény 7.0



Meg tudod csinálni.

Vízszintes

1. A művész homokozója (angolul)
4. A keretes elrendezésben a maradék helye
5. Java-alapstílus
9. Vár és figyel
11. Esemény, angolul
12. Vezérlőt alkalmaz (angolul)
15. A JPanel alapértelmezett elrendezése
16. Többalakúsági teszt
17. Ez mozog! (angolul)
21. Sok mondanivalója van
23. Többet is bekapcsolhatsz (angolul)
25. A gomb barátja
26. Az actionPerformed otthona

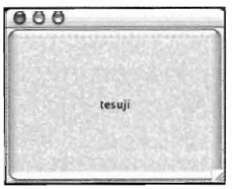
Függőleges

2. A Swing szülője
3. A keretek beállítófüggvénye
5. A File vagy a Help is ez (angolul)
6. Szórakoztatóbb, mint a szöveg (angolul)
7. Összetevő, programozói szlengben (és angolul)
8. Ki- vagy bekapcsolja az álcázóberendezést az űrhajón
9. Elrendezés, angolul
10. A keretes elrendezés felső része
13. A kezelő szabályainak összessége (angolul)
14. Ezt csinálja a forrás (angolul)
15. Alapértelmezésben keretes elrendezésű
18. Ezt csinálja a felhasználó (angolul)
19. Ez veszi körül a belsőt (angolul)
20. Háttérösszetevő
22. Mac-alapstílus
24. A keretes elrendezés jobb oldala



Gyakorlat – Megoldás

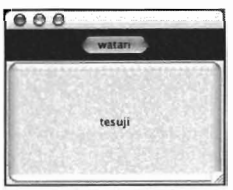
1



C

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

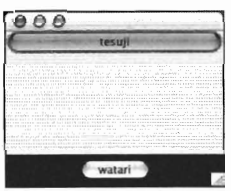
2



D

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.NORTH,panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

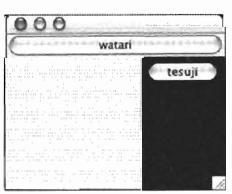
3



E

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.SOUTH,panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.NORTH,button);
```

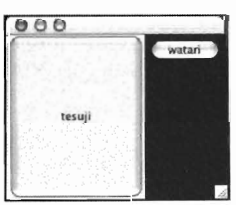
4



A

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(button);
frame.getContentPane().add(BorderLayout.NORTH,buttonTwo);
frame.getContentPane().add(BorderLayout.EAST, panel);
```

6

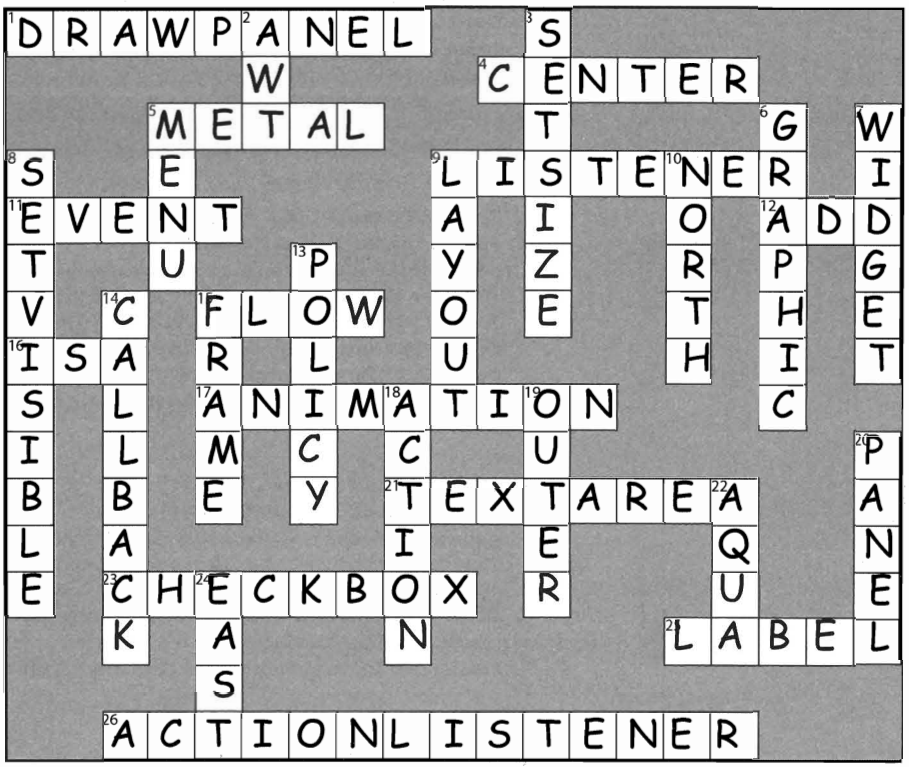


B

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
frame.getContentPane().add(BorderLayout.EAST, panel);
```



GUI-keresztrejtvény 7.0 – megoldás



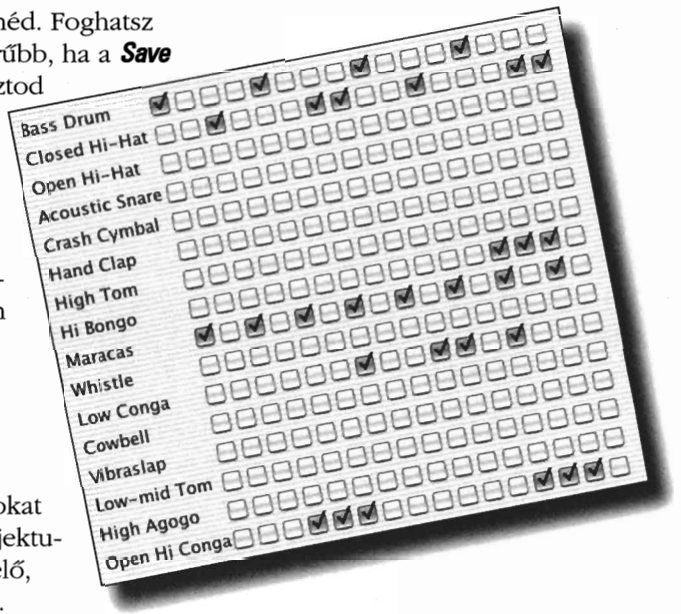
Objektumok mentése



Az objektumok kilapíthatók és felfújhatók. Az objektumoknak állapotuk (state) és viselkedésük (behavior) van. A *viselkedés* helye az osztály, az *állapotot* azonban az egyes *objektumok* tárolják. Mit tehetsz hát, ha *mentened* kell egy objektum állapotát? Ha egy játékot írsz, szükséged lesz egy Save/Restore Game (Játék mentése/visszaállítása) szolgáltatásra. Ha olyan alkalmazást készítesz, amely grafikonokat rajzol, el kell látnod egy Save/Open File (Fájl mentése/megnyitása) lehetőséggel. Ha a programodnak állapotot kell mentenie, *csinálhatod bonyolultan*, kikérdezve minden objektumot, majd fáradságos munkával egy általad létrehozott formátumú fájlba írva minden egyes példányváltozó értékét – vagy a könnyű objektumközpontú módon: egyszerűen lefagyasztva/kilapítva/elraktározva/kiszárítva magát az objektumot, majd felolvasztva/felfújva/visszaállítva/újrahidratálva azt. Mindazonáltal *néha* muszáj a nehezebb utat választani, különösen ha az alkalmazásod által mentett fájlt egy másik, nem Java nyelvű alkalmazásnak kell majd visszaolvasnia, ezért ebben a fejezetben mindkét módszerrel megismertetünk.

Kapd el a ritmust!

Elkészítetted a tökéletes dobmintát, és *menteni* szeretnéd. Foghatsz egy darab papírt, és nekiállhatsz körmölni, de egyszerűbb, ha a ***Save*** (Mentés) gombra kattintasz (vagy a Save pontot választod a File menüből). Ez után nevet adhatsz a mintának, kiválaszthatsz egy könyvtárat, és megkönnyebbülten felsóhajthatsz, tudván, hogy a mesterműved nem enyészik el a semmiben, amikor beáll a kék halál. Egy Java-program állapotát sokféleképpen mentheted – a választás valószínűleg attól függ, hogy mire szeretnéd *használni* a mentett állapotot. Ebben a fejezetben a következő lehetőségeket tekintjük át:



Ha az adataidat csak az azokat előállító Java-program fogja használni:

① **Alkalmazz sorosítást**

Írj egy fájlt, amely kilapított (sorosított) objektumokat tárol. A programoddal olvastasd ki a sorosított objektumokat ebből a fájlból, és fújd fel őket, hogy újra élő, lélegző, kupaclakó objektumok legyenek belőlük.

Ha az adataidat más programok is használni fogják:

② **Írj egy sima szövegfájlt**

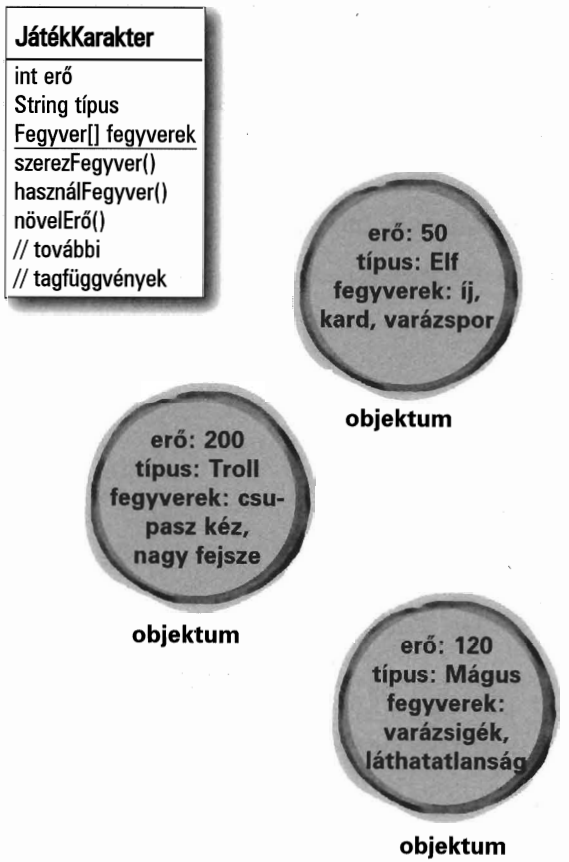
Írj egy fájlt, olyan határolójelekkel, amelyeket más programok is tudnak értelmezni. Egy tabulátorokkal tagolt fájlt például a táblázat- és adatbázis-kezelő programok is fel tudnak dolgozni.

Természetesen egyéb lehetőségeid is vannak – az adatokat bármilyen általad választott formátumban mentheted, például karakterek helyett bájtok formájában is felírhatod az adataidat. Vagy kiírhatsz bármilyen Java-primitívet *Java-primitívként* – léteznek tagfüggvények int-ek, long-ok, boolean-ok stb. kiírására. A használt tagfüggvénytől függetlenül az alapvető írási-olvasási (bemeneti-kimeneti I/O) módszerek lényegében azonosak: adatokat írsz *valamibe*, és az a valami általában vagy egy fájl a lemezen, vagy egy hálózati kapcsolatról érkező adatfolyam (stream). Az adatok visszaolvasása ugyanígy zajlik, csak fordítva: adatokat olvasol be vagy egy lemezen levő fájlból, vagy egy hálózati kapcsolaton keresztül. Természetesen minden, amiről most szó van, akkor érvényes, ha nem egy valódi adatbázist használsz.

Az állapot mentése

Tegyük fel, hogy van egy fantasy-játékprogramod, amelyet nem lehet egyszerre végigjátszani. A játék folyamán a játék karakterei erősebbek vagy gyengébbek, ügyesebbek vagy ügyetlenebbek lesznek, fegyvereket szereznek és vesztenek el, és így tovább. Nem szeretnéd, ha a játékot mindig elölről kellene kezdeni, amikor elindítod a programot, hiszen nem kevés időbe telik a karaktereket csúcsformába hozni egy látványos csatához. Kell hogy legyen tehát valamilyen mód a karakterek állapotának mentésére, illetve a játék folytatásakor az állapot visszaállítására. Mivel pedig nem csak játékos vagy, hanem maga a programozó is, azt szeretnéd, ha a mentés és visszaállítás a lehető legegyszerűbb (és a lehető „legbolondbiztosabb") lenne.

Tegyük fel, hogy három játékkaraktert kell mentened...



① Első lehetőség

A három sorosított karakterobjektumot fájlba írod

Létrehozol egy fájlt, és beleírod a három sorosított karakterobjektumot. A fájl tartalma szövegként olvasva értelmetlen lesz:

```
"IsrJátékKarakter
"%ge8MUIerőLjava/lang/
String;[fegyverek[Ljava/lang/
String;xp2tlfur[Ljava.lang.String;?"VÁ
E{Gxptíjtkardtvarázsporsq~"tTrolluq~tcs
upasz kézstnagy fejszesq~xtMágusuq~tvará
zsigéktláthatatlanság
```

② Második lehetőség

Az adatokat egy sima szövegfájlba írod

Létrehozol egy fájlt, és három szövegsort írsz bele – egyet-egyet a három karakter számára –, az állapotjellemzőket vesszőkkel elválasztva:

```
50,Elf,íj,kard,varázspor
200,Troll,csupasz kéz,nagy fejsze
120,Mágus,varázsigék,láthatatlanság
```

A sorosított fájl az emberi szemnek sokkal nehezebben olvasható, de a programod jóval egyszerűbben (és biztonságosabban) képes visszaállítani a három objektumot a sorosított állapotból, mint ha azok változóinak értékét egy szövegfájlba mentenéd, és onnan olvasnád vissza. Képzeld el például, hogy véletlenül rossz sorrendben olvasod vissza az értékeket – a karakter típusa „varázspor" lesz „Elf" helyett, míg az „Elf"-ből fegyver válik...

Sorosított objektum fájlba írása

Az alábbiakban láthatod azokat a lépéseket, amelyek egy objektum sorosításához (serialization) vagy más szóval mentéséhez szükségesek. Ne fáradj a lépések memorizálásával – a fejezet későbbi részében alaposabban is elmélyedünk a részletekben.

1 Létrehozol egy fájlkimeneti adatfolyamot (FileOutputStream)

```
FileOutputStream fileStream = new FileOutputStream("Jatek.ser");
```

Ha a „Jatek.ser" fájl nem létezik, automatikusan létrejön.

Létrehozunk egy FileOutputStream objektumot. A FileOutputStream tudja, hogyan kapcsolódjon egy fájlhoz (és hogyan hozza létre azt).

2 Létrehozol egy objektumkimeneti adatfolyamot (ObjectOutputStream)

```
ObjectOutputStream os = new ObjectOutputStream(fileStream);
```

Az ObjectOutputStream lehetővé teszi objektumok kiírását, de nem képes közvetlenül kapcsolódni egy fájlhoz – bele kell táplálni egy „segédet". Ezt hívják az adatfolyamok egymáshoz „láncolásának" (chaining).

3 Kiírod az objektumot

```
os.writeObject(characterOne);
os.writeObject(characterTwo);
os.writeObject(characterThree);
```

Ezek sorosítják a karakterEgy, karakterKetto és karakterHarom által hivatkozott objektumokat, és írják azokat a „Jatek.ser" fájlba.

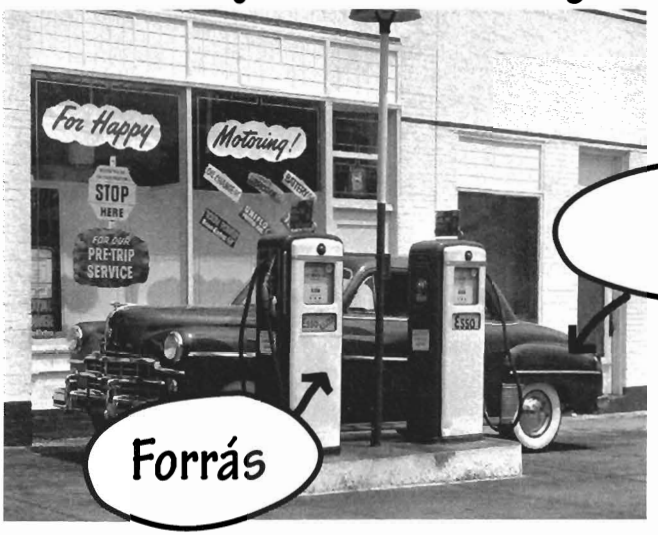
```
os.writeObject(karakterEgy);
os.writeObject(karakterKetto);
os.writeObject(karakterHarom);
```

4 Bezárod az objektumkimeneti adatfolyamot

```
os.close();
```

A felső adatfolyam bezárása bezárja az alatta levőket is, így a FileOutputStream (és a fájl) automatikusan bezáródik.

Az adatok folyamokban mozognak egyik helyről a másikra



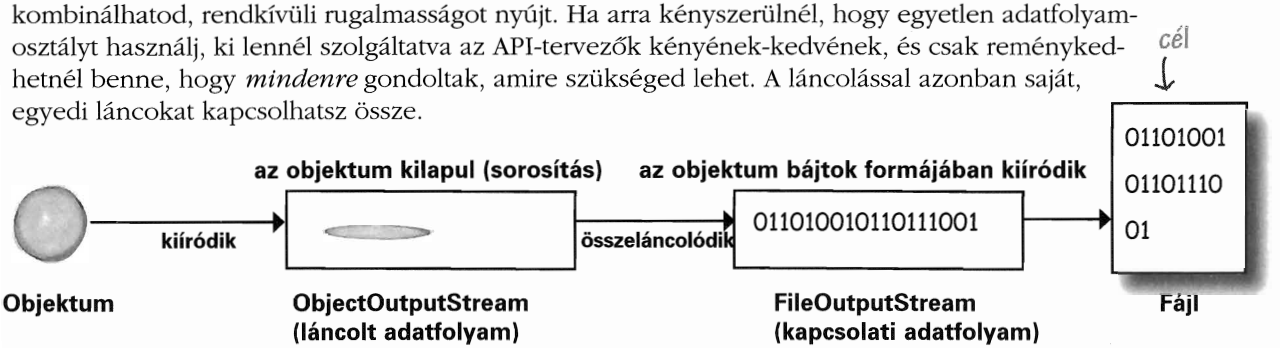
A kapcsolati adatfolyamok (connection stream) egy forráshoz vagy célhoz (fájl, csatoló stb.) vezető kapcsolatot jelképeznek, míg a láncolt adatfolyamok (chain stream) önmaguk nem tudnak kapcsolódni, ezért egy kapcsolati folyamhoz kell láncolni őket.

A Java I/O API-ban ***kapcsolati*** adatfolyamok találhatók, amelyek az olyan forrásokhoz és célokhoz vezető kapcsolatokat jelképeznek, mint a fájlok vagy a hálózati csatolók, illetve ***láncolt*** adatfolyamok, amelyek csak akkor működnek, ha más adatfolyamokhoz láncolod őket.

Gyakran legalább két egymáshoz kapcsolt adatfolyam szükséges ahhoz, hogy valami hasznosat csinálhass: az *egyik* a kapcsolatot jelképezi, míg a *másikra* tagfüggvényeket hívsz meg. Miért kell kettő? Azért, mert a *kapcsolati* adatfolyamok általában túl alacsonyszintűek. A FileOutputStream (fájlkimeneti adatfolyam – ez egy kapcsolati adatfolyam) például *bájtok* kiírására való tagfüggvényekkel rendelkezik, mi viszont nem *bájtokat* akarunk kiírni, hanem *objektumokat*, ezért magasabb szintű *láncolt* adatfolyamra van szükségünk.

Jó, de akkor miért nem egyetlen adatfolyamot használunk, ami *pontosan* azt csinálja, amit akarunk? Egy olyan adatfolyamot, amely lehetővé teszi, hogy objektumokat írjunk ki, és a háttérben alakítja azokat bájtokká? Gondolj a helyes objektumközpontú tervezés elveire: minden osztálynak *egyetlen* dolgot kell jól csinálnia – a FileOutputStream-ek bájtokat írnak egy fájlba, az ObjectOutputStream-ek (objektumkimeneti adatfolyamok) pedig objektumokat alakítanak olyan adatokká, amelyeket adatfolyamba lehet írni. Ezért hát egy FileOutputStream-et hozunk létre, amellyel fájlba írhatunk, és a végéhez hozzáláncolunk egy ObjectOutputStream-et (egy láncolt adatfolyamot). Amikor meghívjuk a writeObject() tagfüggvényt az ObjectOutputStream-re, az objektumot az adatfolyamba pumpáljuk, ahonnan az a FileOutputStream-be kerül, ahol végül bájtok formájában egy fájlba íródik.

Az a lehetőség, hogy a kapcsolati folyamokat és a láncolt adatfolyamokat különböző módokon kombinálhatod, rendkívüli rugalmasságot nyújt. Ha arra kényszerülnél, hogy egyetlen adatfolyamosztályt használj, ki lennél szolgáltatva az API-tervezők kényének-kedvének, és csak reménykedhetnél benne, hogy *mindenre* gondoltak, amire szükséged lehet. A láncolással azonban saját, egyedi láncokat kapcsolhatsz össze.



Valójában mi történik egy objektummal, amikor sorosítod?

1 Objektum a kupacban



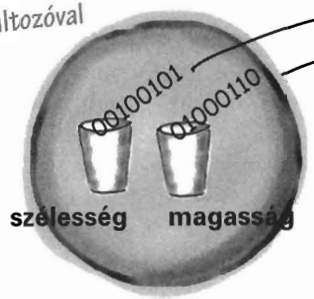
A kupacban levő objektumoknak állapotuk van – ez az objektum példányváltozóiban tárolt értékek összességét jelenti. Ezek az értékek teszik egymástól különbözővé egy osztály példányait.

2 Sorosított objektum

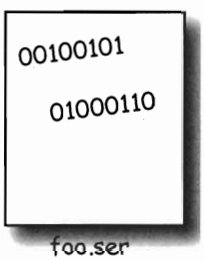


A sorosított objektumok **mentik a példányváltozók értékét**, hogy később egy pontosan ugyanolyan példányt (objektumot) kelthess életre a kupacban.

Objektum két primitív példányváltozóval



Az értékeket kiszippantjuk, és belepumpáljuk az adatfolyamba.



A szélesség és magasság példányváltozók értékét a „foo.ser" fájlba mentjük néhány kiegészítő információval együtt, amelyre a JVM-nek szüksége van ahhoz, hogy visszaállíthassa az objektumot (ilyen például az objektum osztálytípusa).

```
Foo myFoo = new Foo();
myFoo.setWidth(37);
myFoo.setHeight(70);
```

```
FileOutputStream fs = new FileOutputStream("foo.ser");
ObjectOutputStream os = new ObjectOutputStream(fs);
os.writeObject(myFoo);
```

Létrehozunk egy FileOutputStream-et, amely a „foo.ser" fájlhoz kapcsolódik, majd hozzálánco-lunk egy ObjectOutputStream-et, és utasítjuk az utóbbit, hogy írja ki az objektumot.

De pontosan miből is áll egy objektum állapota? Mit kell menteni?

A dolog most kezd érdekessé válni. Az olyan *primitív* értékeket, mint a 37 vagy a 70, könnyű menteni. De mi van, ha az objektum egyik példányváltozója egy objektum*hivatkozás*? És ha az objektumnak öt ilyen példányváltozója is van? És ha ezeknek az objektum típusú példányváltozóknak maguknak is vannak példányváltozóik? Gondold végig. Egy objektumnak milyen részei lehetnek egyediek? Képzeld magad elé, hogy mit kell visszaállítani ahhoz, hogy egy pontosan ugyanolyan objektumot kapj, mint amilyet mentettél. A memóriában elfoglalt helye természetesen más lesz, de ez nem számít. Csak az számít, hogy a kupacba egy olyan objektum kerül, amelynek az állapota megegyezik azzal az állapottal, amelyet mentettél.



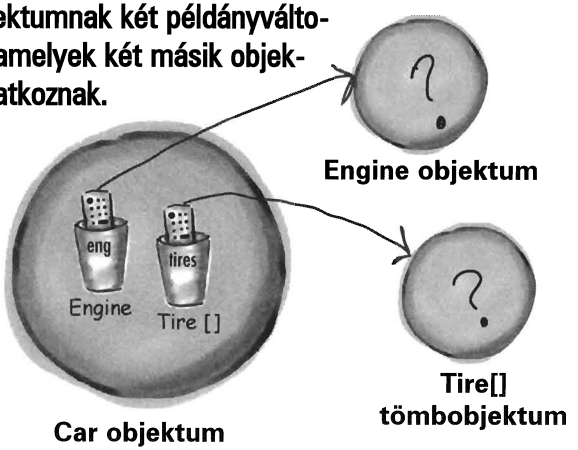
Agygyúró

Mit kell tenned ahhoz, hogy a Car (Autó) objektumot úgy mentsd, hogy visszaállítható legyen az eredeti állapotába?

Gondold végig, mit és hogyan kell menteni egy autó elraktározásához.

Mi történik, ha egy Engine (Motor) objektum egy Carburator (Karburátor) objektumra hivatkozik? És mi van a Tire[] (Abroncs) tömbobjektumon belül?

A Car objektumnak két példányváltozója van, amelyek két másik objektumra hivatkoznak.



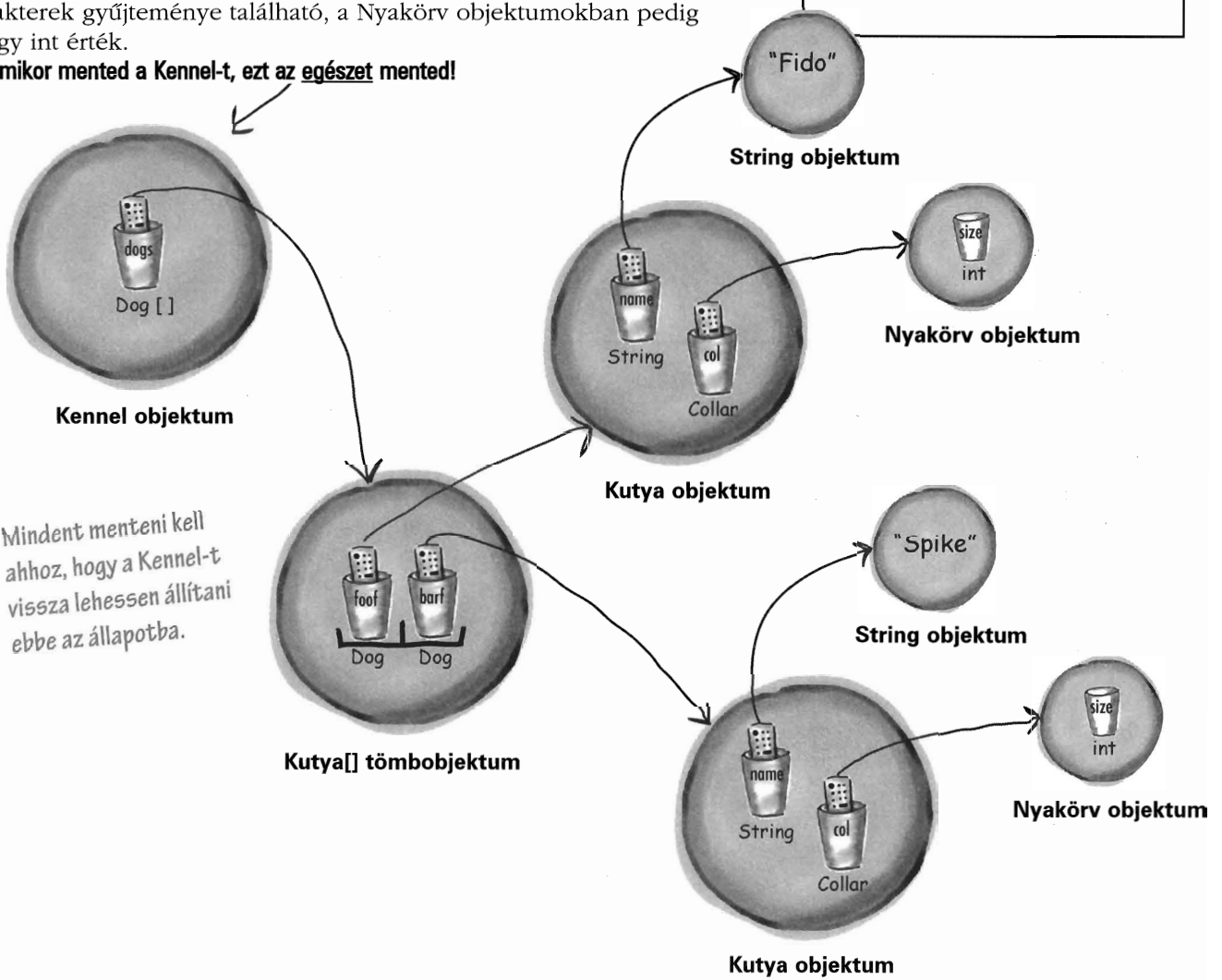
Mit kell tenni egy Car objektum mentéséhez?

Amikor sorosítasz egy objektumot, a példányváltozóiból hivatkozott valamennyi objektumot szintén sorosítod, és azokat az objektumokat is, amelyekre ezek az objektumok hivatkoznak. És azokat az objektumokat is, amelyekre ez utóbbi objektumok hivatkoznak… De a legjobb az, hogy ez az egész automatikusan történik!

Az alábbi Kennel objektum egy Kutya[] tömbobjektumra hivatkozik. A Kutya[] tömb két Kutya objektum hivatkozását tartalmazza, az egyes Kutya objektumok pedig egy String és egy Nyakörv objektumra mutató hivatkozást tárolnak. A String objektumokban karakterek gyűjteménye található, a Nyakörv objektumokban pedig egy int érték.

A sorosítás a teljes objektumgráfot menti, vagyis a példányváltozók által hivatkozott összes objektumot, a sorosított objektumtól kezdve.

Amikor mented a Kennel-t, ezt az egészet mented!



Ha azt szeretnéd, hogy egy osztály sorosítható legyen, meg kell valósítanod a Serializable felületet

A Serializable („sorosítható") felület egyfajta jelölőfelület (marker) vagy címkefelület (tag), mert a felület nem rendelkezik egyetlen megvalósítandó tagfüggvénnyel sem. Az egyetlen célja az, hogy bejelentse, hogy az őt megvalósító osztály *sorosítható*, vagyis hogy az ilyen típusú objektumok menthetők sorosításon keresztül. Ha egy osztály bármely ősosztálya sorosítható, az alosztály is automatikusan sorosítható lesz, akkor is, ha az alosztály nem jelenti be nyíltan, hogy megvalósítja a Serializable felületet (*implements Serializable*). (A felületek mindig így működnek: ha az alaposztályod „EGYFAJTA" (IS-A) Serializable, akkor te is az vagy.)

```
objectOutputStream.writeObject(myBox);
```

Akármi is kerüljön ide, meg KELL valósítania a Serializable felületet, különben futáskor kudarcot vall.

```
import java.io.*;
```

A Serializable felület a java.io csomagban található, ezért be kell tölteni.

```
public class Box implements Serializable {
```

Nincsenek megvalósítandó tagfüggvények, de amikor azt mondod, hogy „implements Serializable", az azt közli a JVM-mel, hogy az ilyen típusú objektumok nyugodtan sorosíthatók.

```
    private int width;
    private int height;
```

Ezt a két értéket mentjük.

```
    public void setWidth(int w) {
       width = w;
    }

    public void setHeight(int h) {
       height = h;
    }

public static void main (String[] args) {

      Box myBox = new Box();
      myBox.setWidth(50);
      myBox.setHeight(20);

      try {
         FileOutputStream fs = new FileOutputStream("foo.ser");
         ObjectOutputStream os = new ObjectOutputStream(fs);
         os.writeObject(myBox);
         os.close();
      } catch(Exception ex) {
          ex.printStackTrace();
      }
   }
}
```

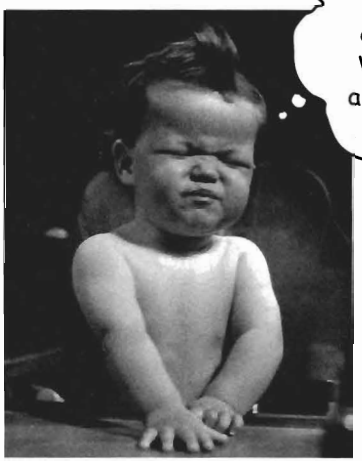
Az I/O-műveletek kivételeket válthatnak ki.

Kapcsolódunk a „foo.ser" nevű fájlhoz, ha az létezik. Ha nem, létrehozunk egy új fájlt „foo.ser" néven.

Létrehozunk egy ObjectOutputStream-et a kapcsolati adatfolyamhoz láncolva, és arra utasítjuk, hogy írja ki az objektumot.

A sorosítás a „mindent vagy semmit" elvén működik.

El tudod képzelni, mi történne, ha az objektum állapotának egy részét nem sikerül helyesen menteni?



Fúj! Rosszul leszek, ha csak rá gondolok! Milyen már, ha mondjuk egy súly nélküli Kutyát kapok vissza? Vagy egy fületlent? Vagy ha mondjuk a nyakörve 3-as méretű lesz 30-as helyett? Ezt nem szabad megengedni!

Ha nem sikerül helyesen sorosítani a teljes objektumgráfot, a sorosítás meghiúsul. Nem sorosíthatsz egy Tó objektumot, ha a Kacsa példányváltozója nem hajlandó alávetni magát a sorosításnak (mert nem valósítja meg a Serializable felületet).

```
import java.io.*;

public class Pond implements Serializable {

    private Duck duck = new Duck();

    public static void main (String[] args) {
       Pond myPond = new Pond();
       try {
         FileOutputStream fs = new FileOutputStream("Pond.ser");
         ObjectOutputStream os = new ObjectOutputStream(fs);

         os.writeObject(myPond);
         os.close();

       } catch(Exception ex) {
          ex.printStackTrace();
       }
    }
}

public class Duck {
   // ide jön a kacsa kódja
}
```

A Pond (Tó) objektumok sorosíthatók.

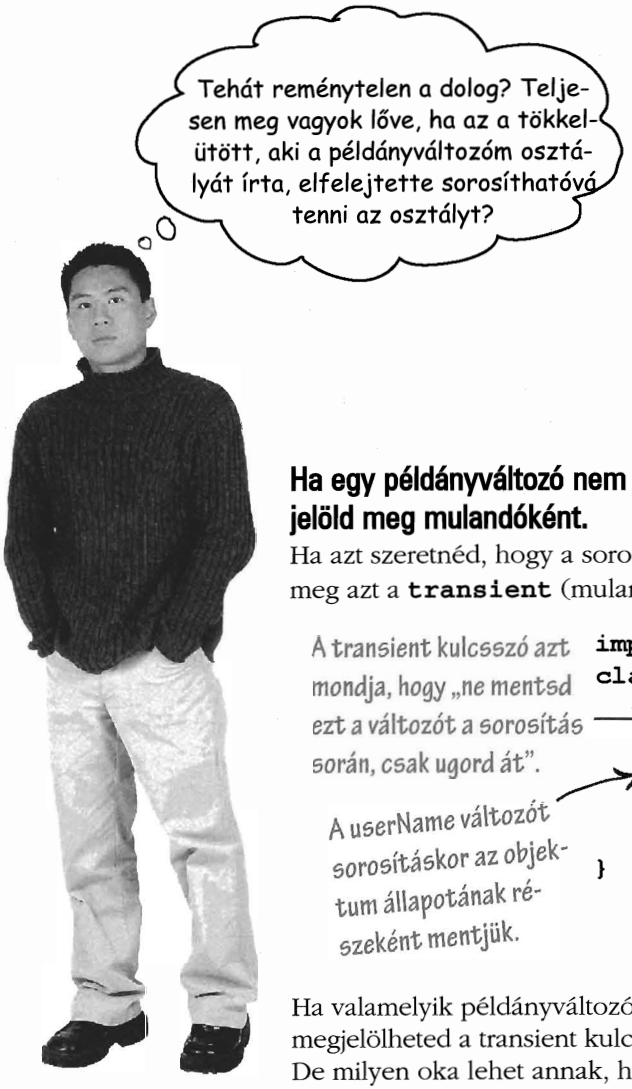
A Pond osztály egyetlen, Duck (Kacsa) típusú példányváltozóval rendelkezik.

Amikor sorosítod a myPond-ot (ami egy Pond típusú objektum), a Duck példányváltozója is automatikusan sorosított lesz.

Jaj! A Duck nem sorosítható! Nem valósítja meg a Serializable felületet, ezért amikor megpróbálsz sorosítani egy Pond objektumot, kudarcot vallasz, mert a Pond Duck példányváltozója nem menthető.

Ezt kapod, amikor megpróbálod futtatni a Pond osztály main tagfüggvényét:

```
File Edit Window Help Regret
% java Pond
java.io.NotSerializableException: Duck
        at Pond.main(Pond.java:13)
```



Ha egy példányváltozó nem menthető (vagy nem szabad menteni), jelöld meg mulandóként.

Ha azt szeretnéd, hogy a sorosító eljárás átugorjon egy példányváltozót, jelöld meg azt a **`transient`** (mulandó) kulcsszóval.

A transient kulcsszó azt mondja, hogy „ne mentsd ezt a változót a sorosítás során, csak ugord át”.

A userName változót sorosításkor az objektum állapotának részeként mentjük.

```
import java.net.*;
class Chat implements Serializable {
  transient String currentID;
  String userName;
  // további kód
}
```

Ha valamelyik példányváltozódat nem lehet menteni, mert nem sorosítható, megjelölheted a transient kulcsszóval, és a sorosító eljárás egyszerűen átugorja. De milyen oka lehet annak, ha egy változó nem sorosítható? Nos, lehet, hogy az osztály tervezője egyszerűen elfelejtette megvalósíttatni az osztállyal a Serializable felületet. Az is lehet, hogy az objektum olyan futásidejű információkra támaszkodik, amiket nem lehet menteni. Bár a Java-osztálykönyvtárakban található dolgok többsége sorosítható, az olyan dolgokat nem mentheted, mint a hálózati kapcsolatok, a szálak vagy a fájlobjektumok, mert ezek mindegyike egy adott futásidejű „élménytől” függ – más szavakkal, a példányosításuk olyan módon történik, ami egyedi a program adott futtatására, illetve a használt rendszerre vagy JVM-re nézve. Miután a program befejezte a futását, nincs rá mód, hogy értelmes módon visszaállítsd ezeket a dolgokat, mert minden alkalommal teljesen újat kell létrehozni belőlük.

Nincsenek
hülye kérdések

K: Ha a sorosítás ilyen fontos, miért nem alapértelmezett minden osztály számára? Miért nem valósítja meg az Object a Serializable felületet, hogy minden alosztálya automatikusan sorosítható legyen?

V: Bár a legtöbb osztálynak szabad és célszerű is megvalósítania a Serializable felületet, így mindig van választási lehetőséged. Neked pedig tudatos döntést kell hoznod minden egyes általad tervezett osztály esetében, hogy „engedélyezd" a sorosítást a Serializable felület megvalósításával. Először is, ha a sorosíthatóság alapértelmezett lenne, hogyan kapcsolnád ki? A felületek valamilyen szolgáltatást jeleznek, nem egy szolgáltatás *hiányát*, ezért a többalakúság nem működne helyesen, ha azt kellene mondanod, hogy „implements NonSerializable", hogy tudasd a világgal, hogy az adott osztály nem menthető.

K: Mi célból írnék olyan osztályt, ami *nem* sorosítható?

V: Erre kevés okod lehet, de előfordulhat például, hogy a biztonságot kell szem előtt tartanod, és nem szeretnél elraktározni egy jelszóobjektumot. Az is lehet, hogy az egyik objektumodat nincs értelme menteni, mert a kulcsfontosságú példányváltozói maguk nem sorosíthatók, ezért nem veszed hasznát az *osztályod* sorosításának.

K: Ha egy osztály nem sorosítható, pedig ezt nem igazán indokolja semmi (csak éppen a tervezője megfeledkezett a sorosításról, vagy ostoba volt), származtathatok sorosítható *alosztályt* a „rossz" osztályból?

V: Igen! Ha maga az osztály bővíthető (vagyis nem „final"), származtathatsz belőle sorosítható alosztályt, és behelyettesítheted azt bárhová, ahol a kód az alaposztály típusát várja. (Emlékezz vissza, hogy a többalakúság ezt megengedi.) Ez viszont felvet egy érdekes kérdést: mit *jelent* egyáltalán, ha az alaposztály nem sorosítható?

K: Ha már így felhoztad a kérdést: mit *jelent* az, ha egy nem sorosítható alaposztálynak sorosítható alosztálya van?

V: Először meg kell még tanulnod, mi történik, amikor egy sorosított osztályt visszafejtesz (erről a következő néhány oldalon lesz szó). Dióhéjban, ha egy objektumot visszafejtesz, és az alaposztálya *nem* sorosítható, az alaposztály konstruktora úgy fog lefutni, mintha új objektumot hoznál létre az adott típusból. Ha nincs ésszerű oka annak, hogy egy osztály nem sorosítható, egy sorosítható alosztály létrehozása jó megoldás lehet.

K: Fú! Most rájöttem valami fontosra... Ha egy változót „mulandóvá" teszünk, az azt jelenti, hogy a sorosítás átugorja a változó értékét. De mi történik vele utána? A nem sorosítható példányváltozók problémáját megoldjuk azzal, hogy a példányváltozót mulandóvá tesszük, de nem lesz SZÜKSÉG erre a változóra, amikor az objektumot ismét életre keltjük? Más szavakkal, a sorosításnak nem az lenne a lényege, hogy megőrizzük egy objektum állapotát?

V: Igen, ez fontos probléma, de szerencsére van megoldás. Amikor sorosítasz egy objektumot, a mulandó hivatkozó példányváltozókat *null*-ként kapod vissza, függetlenül attól, hogy a mentés idején milyen érték volt bennük. Ez azt jelenti, hogy a sorosítás az adott példányváltozóhoz kapcsolódó teljes objektumgráfot átugorja, vagyis nem menti. Ez nyilvánvalóan nem kívánatos, mert valószínűleg nem null értéket szeretnél a kérdéses változó számára.

Két lehetőséged van:

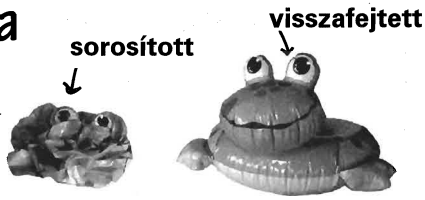
1) Amikor az objektumot ismét életre kelted, az üres példányváltozó értékét visszaállítod valamilyen alapértelmezett állapotba. Ez akkor működik, ha a visszafejtett objektum nem függ az adott mulandó változó értékétől. Fontos lehet például, hogy egy Kutyának Nyakörve legyen, de ha minden Nyakörv objektum egyforma, akkor nem számít, ha a feltámasztott Kutyának vadonatúj Nyakörvet adsz – senki sem fogja észrevenni a különbséget.

2) Ha viszont a mulandó változó értéke *igenis* számít (mert mondjuk minden Kutyának egyedi színű és tervezésű mulandó Nyakörve van), akkor mentened kell a Nyakörv kulcsfontosságú értékeit, és fel kell használnod azokat, amikor a Kutyát feltámasztod, hogy lényegében az eredetivel megegyező vadonatúj Nyakörvet adj neki.

K: Mi történik, ha az objektumgráf két objektuma ugyanaz az objektum – Például ha két különböző Cat (Macska) objektum van a Kennel-ben, de mindkettő ugyanarra az Owner (Tulajdonos) objektumra hivatkozik? Az Owner mentésére kétszer kerül sor? Remélem, nem.

V: Kitűnő kérdés! A sorosító eljárás elég okos ahhoz, hogy felismerje, ha egy objektumgráf két objektuma azonos. Ebben az esetben csak az *egyik* objektumot menti, visszafejtéskor pedig minden hivatkozást helyreállít, ami erre az egyetlen objektumra mutat.

Visszafejtés: az objektumok visszaállítása



Egy objektum sorosításának az a célja, hogy valamikor később, a JVM egy újabb „futtatásakor" visszaállíthasd az eredeti állapotába. (Még az is lehet, hogy nem ugyanazt a JVM-et fogod futtatni, mint amelyiket akkor használtad, amikor az objektum sorosítására sor került). A sorosítás visszafejtése (deserialization) majdnem ugyanúgy történik, mint a sorosítás, csak visszafelé.

1 Létrehozol egy fájlbemeneti adatfolyamot (FileInputStream)

Ha a „MyGame.ser" fájl nem létezik, kivételt kapsz.

```
FileInputStream fileStream = new FileInputStream("MyGame.ser");
```

Létrehozunk egy FileInputStream objektumot. A FileInputStream tudja, hogyan kapcsolódjon egy meglevő fájlhoz.

2 Létrehozol egy objektumbemeneti adatfolyamot (ObjectInputStream)

```
ObjectInputStream os = new ObjectInputStream(fileStream);
```

Az ObjectInputStream lehetővé teszi az objektumok visszaolvasását, de nem képes közvetlenül kapcsolódni egy fájlhoz – hozzá kell láncolni egy kapcsolati adatfolyamhoz, ebben az esetben egy FileInputStream-hez.

3 Kiolvasod az objektumokat

```
Object one = os.readObject();
Object two = os.readObject();
Object three = os.readObject();
```

Minden alkalommal, amikor azt mondod, hogy readObject(), a következő objektumot kapod meg az adatfolyamból, tehát az objektumokat abban a sorrendben olvasod vissza, ahogy kiírtad őket. Ha több objektumot próbálsz kiolvasni, mint amennyit kiírtál, egy szép kövér kivételt kapsz.

4 Átalakítod az objektumokat

```
GameCharacter elf = (GameCharacter) one;
GameCharacter troll = (GameCharacter) two;
GameCharacter magician = (GameCharacter) three;
```

A readObject() visszatérési típusa Object (mint az ArrayList esetében), ezért vissza kell alakítanod az általad ismert valódi típusára.

5 Bezárod az objektumbemeneti adatfolyamot

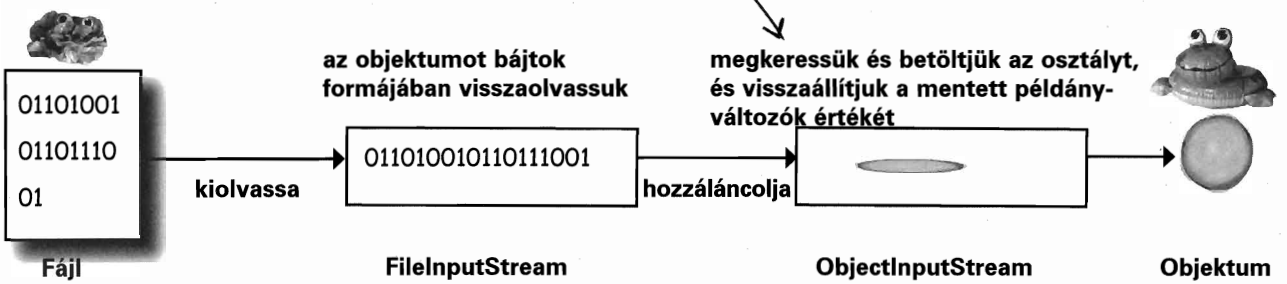
```
os.close();
```

A felső adatfolyam bezárása bezárja az alatta levőket is, így a FileInputStream (és a fájl) automatikusan bezáródik.

Mi történik, amikor visszafejtesz egy objektumot?

Amikor egy sorosított objektumot visszafejtesz, a JVM megkísérli újra életre kelteni azt egy olyan új objektum létrehozásával a kupacban, amelynek az állapota megegyezik az objektum sorosításkori állapotával – persze a mulandó változóktól eltekintve, amelyeket vagy null értékkel kapsz vissza (ha objektumhivatkozásokról van szó), vagy az alapértelmezett primitív értékekkel.

Ez a lépés kivételt vált ki, ha a JVM nem találja vagy nem tudja betölteni az osztályt!



1. Az objektumot **kiolvassuk** az adatfolyamból.

2. A JVM meghatározza az objektum **osztálytípusát** (a sorosított objektummal együtt tárolt információk alapján).

3. A JVM megkísérli **megkeresni és betölteni** az objektum **osztályát**. Ha a JVM nem találja, illetve nem tudja betölteni az osztályt, kivételt vált ki, és a visszafejtés kudarcot vall.

4. Az új objektum helyet kap a kupacban, de a **sorosított objektum konstruktora NEM fut le**! Nyilvánvaló, hogy ha a konstruktor lefutna, az objektum állapotát az eredeti „új" állapotba állítaná visszza, ami nem az, amit akarunk. Mi abban az állapotban akarjuk tudni az objektumot, amelyben akkor volt, *amikor sorosítottuk*, nem pedig akkor, amikor létrehoztuk.

5. Ha az objektum öröklési fájában valahol feljebb található egy nem sorosítható osztály is, **annak a konstruktora** a felette található összes konstruktorral együtt **lefut** (akkor is, ha ez utóbbi osztályok maguk sorosíthatók). Ha a konstruktorok összeláncolása megkezdődött, már nem állíthatod le, ami azt jelenti, hogy az első nem sorosítható osztálytól kezdve minden ősosztály alapállapotba áll vissza.

6. Az objektum példányváltozói visszakapják a sorosított állapotban tárolt értéküket. A mulandó változók null értéket kapnak, ha objektumhivatkozások, illetve az alapértelmezett értéket (0, false stb.), ha primitívek.

Nincsenek hülye kérdések

K: Miért nem menti a Java az osztályt az objektum részeként? Így nem fordulhatna elő, hogy az osztály nem található.

V: Természetesen így is megoldhatták volna a sorosítást, de ez hatalmas többletterhelést és pazarlást jelentett volna. Ezenkívül lehet, hogy nem jelent különösebb nehézséget arra használni a sorosítást, hogy egy helyi merevlemezen található fájlba írj ki objektumokat, a sorosítást azonban arra is alkalmazzák, hogy hálózati kapcsolatokon át küldjenek objektumokat. Ha minden sorosított (elküldhető) objektumhoz mellékelni kellene az osztályt, a sávszélesség sokkal komolyabb problémát jelentene, mint amit egyébként okoz.
A hálózati ávitel céljára sorosított objektumokhoz ugyanakkor *létezik* egy eljárás, amely a sorosított objektumra „rábélyegzi" azt az URL-t, ahol az objektum osztálya található. Ezt a megoldást alkalmazza a Javában az RMI (Remote Method Invocation, távoli eljáráshívás), hogy egy sorosított objektumot elküldhess mondjuk egy tagfüggvény-argumentum részeként, és ha a hívást fogadó JVM nem rendelkezik az osztállyal, az URL segítségével automatikusan lehívhassa és betölthesse azt a hálózatról. (Az RMI-ről a 17. fejezetben lesz szó.)

K: Mi a helyzet a statikus változókkal? A sorosítás azokat is menti?

V: Nem. Emlékezz vissza, hogy a „statikus" azt jelenti, hogy „osztályonként egy", nem pedig „objektumonként egy". A statikus változók mentésére tehát nem kerül sor, ha pedig a sorosított objektumot visszafejted, olyan statikus változók kerülnek bele, amilyenekkel az osztály *éppen* rendelkezik. A tanulság: sorosítható objektumokat ne tegyél függővé dinamikusan változó statikus változóktól, mert nem biztos, hogy ugyanazok lesznek, amikor az objektum ismét életre kel.

A játékkarakterek mentése és visszaállítása

```
import java.io.*;

public class GameSaverTest {
   public static void main(String[] args) {
     GameCharacter one = new GameCharacter(50, "Elf", new String[] {"bow", "sword", "dust"});
     GameCharacter two = new GameCharacter(200, "Troll", new String[] {"bare hands", "big ax"});
     GameCharacter three = new GameCharacter(120, "Magician", new String[] {"spells", "invisibility"});

     // ide képzelj olyan kódot, ami olyan dolgokat csinál a karakterekkel,
     // ami megváltoztathatja az állapotértékeiket

     try {
        ObjectOutputStream os = new ObjectOutputStream(new FileOutputStream("Game.ser"));
        os.writeObject(one);
        os.writeObject(two);
        os.writeObject(three);
        os.close();
     } catch(IOException ex) {
        ex.printStackTrace();
     }
     one = null;
     two = null;
     three = null;

     try {
       ObjectInputStream is = new ObjectInputStream(new FileInputStream("Game.ser"));
        GameCharacter oneRestore = (GameCharacter) is.readObject();
        GameCharacter twoRestore = (GameCharacter) is.readObject();
        GameCharacter threeRestore = (GameCharacter) is.readObject();

       System.out.println("One's type: " + oneRestore.getType());
       System.out.println("Two's type: " + twoRestore.getType());
       System.out.println("Three's type: " + threeRestore.getType());
     } catch(Exception ex) {
        ex.printStackTrace();
     }
   }
```

JatekMentoTeszt

Létrehozunk néhány karaktert...

{"íj", "kard", "varázspor"}
{"csupasz kéz", "nagy fejsze"}
{"varázsigék", "láthatatlanság"}

JatekKarakter

"Jatek.ser"

Null-ra állítjuk őket, így nem érhetjük el az objektumokat a kupacban.

egyVisszaallit
kettoVisszaallit
haromVisszaallit

Most visszaolvassuk őket a fájlból...

Megnézzük, hogy működött-e.

"Egy típusa: "
"Kettő típusa: "
"Három típusa: "

```
File Edit Window Help Resuscitate
% java GameSaver
Elf
Troll
Magician
```

Elf (Tünde)
Troll
Mágus

erő: 50
típus: Elf
fegyverek: íj, kard, varázspor
objektum

erő: 200
típus: Troll
fegyverek: csupasz kéz, nagy fejsze
objektum

erő: 120
típus: Mágus
fegyverek: varázsigék, láthatatlanság
objektum

A JatekKarakter osztály

JatekKarakter

```
import java.io.*;

public class GameCharacter implements Serializable {
   int power;
   String type;
   String[] weapons;

   public GameCharacter(int p, String t, String[] w) {
       power = p;
       type = t;
       weapons = w;
   }

   public int getPower() {
     return power;
   }

   public String getType() {
      return type;
   }

   public String getWeapons() {
       String weaponList = "";

       for (int i = 0; i < weapons.length; i++) {
          weaponList += weapons[i] + " ";
       }
       return weaponList;
   }
}
```

Ez csak egy egyszerű osztály a sorosítás teszteléséhez. Ténylegesen nincs meg a játék – de próbálkozhatsz az elkészítésével.

Objektumsorosítás



Újratöltve

- Egy objektum állapotát az objektum sorosításával mentheted.
- Egy objektum sorosításához egy objektumkimeneti adatfolyamra (ObjectOutputStream) van szükséged (a java.io csomagból).
- Az adatfolyamok kapcsolati vagy láncolt adatfolyamok lehetnek.
- A kapcsolati adatfolyamok egy forráshoz vagy célhoz vezető kapcsolatot jelképeznek – jellemzően egy fájlt, hálózati csatolókapcsolatot vagy a konzolt.
- A láncolt adatfolyamok nem képesek egy forráshoz vagy célhoz kapcsolódni, ezért egy kapcsolati (vagy másfajta) adatfolyamhoz kell láncolni őket.
- Ha egy objektumot fájlba szeretnél sorosítani, egy fájlkimeneti adatfolyamot (FileOutputStream) kell létrehoznod, és hozzá kell láncolnod azt egy objektumkimeneti adatfolyamhoz.
- Egy objektum sorosításához a **`writeObject(`*`azObjektum`*`)`** hívást kell kiadnod az objektumkimeneti adatfolyamra. A fájlkimeneti adatfolyamra nem kell tagfüggvényeket meghívnod.
- Ahhoz, hogy egy objektum sorosítható legyen, meg kell valósítania a Serializable felületet. Ha az osztály valamelyik ősosztálya megvalósítja a Serializable felületet, az alosztály is automatikusan sorosítható lesz, akkor is, ha kifejezetten nem szerepel benne az *implements Serializable* kitétel.
- Amikor egy objektumot sorosítasz, a Java a teljes objektumgráfját sorosítja. Ez azt jelenti, hogy a Java a sorosított objektum példányváltozói által hivatkozott valamennyi objektumot és az azok által hivatkozott objektumokat stb. is sorosítja.
- Ha a gráfban bármelyik objektum nem sorosítható, futáskor kivétel lép fel, hacsak át nem ugrod az objektumra hivatkozó példányváltozót.
- Ha azt szeretnéd, hogy a sorosítás átugorjon egy példányváltozót, jelöld meg a *transient* kulcsszóval. A változó értéke a visszafejtés után null lesz (ha objektumhivatkozásról van szó), vagy az alapértelmezett érték (a primitívek esetében).
- A sorosítás során a gráfban található valamennyi objektum osztályának elérhetőnek kell lennie a JVM számára.
- Az objektumok visszaolvasása – a readObject() tagfüggvénnyel – ugyanabban a sorrendben történik, ahogy eredetileg kiírtad azokat.
- A readObject() visszatérési típusa Object, ezért a visszafejtett sorosított objektumokat át kell alakítanod a valódi típusukra.
- A statikus változók sorosítására nem kerül sor! Egy statikus változó értékét nincs értelme egy adott objektum állapotának részeként menteni, mivel az adott típusba tartozó összes objektum egyetlen értéken – az osztályban találhatón – osztozik.

Karakterlánc kiírása szövegfájlba

Objektumokat menteni sorosításon keresztül a legegyszerűbb módja annak, hogy adatokat ments és állíts vissza egy Java-program futtatásai között. Néha azonban az adatokat egy régimódi szövegfájlba szükséges mentened. Tegyük fel például, hogy a Java-programodnak egy olyan egyszerű szövegfájlba kell írnia az adatokat, amelyet egy másik (esetleg nem Java nyelvű) programnak kell majd elolvasnia. Lehet például egy kiszolgálóoldali kisalkalmazásod (servlet; ez egy olyan Java-kód, ami a webkiszolgálón fut), amely a felhasználó által a böngészőben beírt űrlapadatokat vár, és egy olyan szövegfájlba írja azokat, amelyet valaki más majd egy táblázatkezelőbe fog betölteni, hogy elemezze.
A szöveges adatok (valójában karakterláncok, tehát String-ek) kiírása hasonló az objektumok kiírásához, azzal az eltéréssel, hogy objektum helyett String-et írsz ki, és fájlkimeneti adatfolyam (FileOutputStream) helyett fájlírót (FileWriter) használsz (és nem láncolod azt egy objektumkimeneti adatfolyamhoz).

Így néznének ki a játékkarakterek adatai, ha emberi olvasásra szánt szövegfájlba írnád ki azokat.

```
50,Elf,íj,kard,varázspor
200,Troll,csupasz kéz,
nagy fejsze
120,Mágus,varázsigék,
láthatatlanság
```

Sorosított objektum kiírása:

```
objectOutputStream.writeObject(valamilyenObjektum);
```

Karakterlánc kiírása:

```
fileWriter.write("Az első mentett karakterláncom");
```

```
import java.io.*;                          // A FileWriter-hez a java.io csomagra van szükség.

class WriteAFile {
    public static void main (String[] args) {

        try {
            FileWriter writer = new FileWriter("Foo.txt");   // Ha a „Foo.txt" fájl nem létezik, a FileWriter létrehozza.

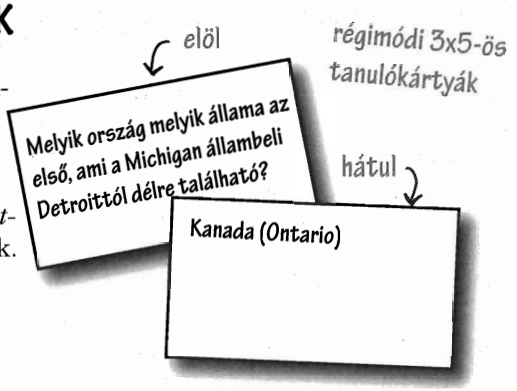
            writer.write("hello foo!");    // A write() tagfüggvény egy karakterláncot vár.

            writer.close();                // Zárd be, ha kész vagy!

        } catch(IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Minden bemeneti-kimeneti műveletet egy try/catch blokkban kell elhelyezned, mert bármi kiválthat IOException-t!!!

Példa szövegfájlra: e-tanulókártyák



Emlékszel még az iskolában használt tanulókártyákra, amelyeknek az egyik oldalán egy kérdést olvashattál, a választ pedig a hátoldalán találtad? Az ilyen kártyák nem segítenek sokat, ha megérteni próbálsz valamit, de a nyers gyakorlásban, amikor csak memorizálni akarsz dolgokat – *amikor tényeket kell beégetned* –, felülmúlhatatlanok. És persze kvízjátékokhoz is kitűnőek.

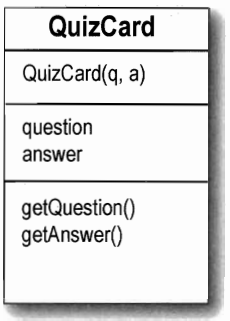
Egy elektronikus változatot fogunk készíteni, amelyben három osztály található:

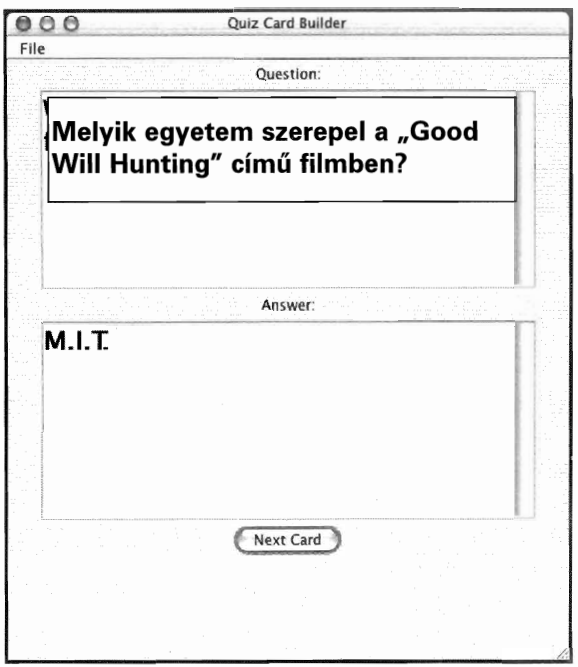
1) ***QuizCardBuilder*** – egyszerű építőeszköz egy csomag e-tanulókártya létrehozásához és mentéséhez.

2) ***QuizCardPlayer*** – lejátszómotor, amely képes betölteni egy tanulókártya-csomagot, és végigpörgetni a kérdéseket a felhasználó számára.

3) ***QuizCard*** – egyszerű osztály, amely a kártyaadatokat ábrázolja.

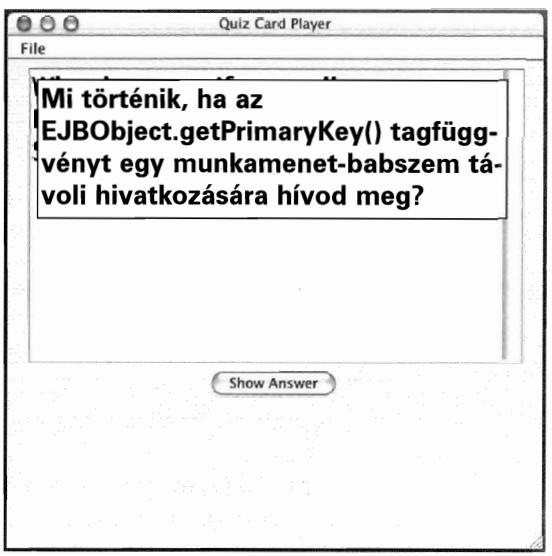
Részletesen megmutatjuk az építő és a lejátszó kódját, hogy a QuizCard osztályt te magad készíthesd el, ennek az alapján:





QuizCardBuilder

Ennek az osztálynak a File, Save (Fájl, Mentés) menüpontja arra szolgál, hogy az éppen összeállított kártyacsomagot egy szövegfájlba mentsd.



QuizCardPlayer

Ennek az osztálynak a File, Load (Fájl, Betöltés) menüpontja arra szolgál, hogy egy kártyacsomagot betölts egy szövegfájlból.

QuizCardBuilder (kódvázlat)

```
public class QuizCardBuilder {

  public void go() {
    // a grafikus felület felépítése és megjelenítése
  }

  Belső osztály
  private class NextCardListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // az aktuális kártya hozzáadása a listához és a szövegmezők kiürítése
    }
  }

  Belső osztály
  private class SaveMenuListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // fájl-párbeszédablak megjelenítése,
      // hogy a felhasználó elnevezhesse és menthesse a csomagot
    }
  }

  Belső osztály
  private class NewMenuListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // a kártyalista és a szövegmezők kiürítése
    }
  }

  private void saveFile(File file) {
    // a kártyalista végigjárása és a kártyák kiírása szövegfájlba
    // feldolgozható módon (vagyis a részeket világosan elválasztva)
  }
}
```

Felépíti és megjeleníti a grafikus felületet, beleértve az eseményfigyelők létrehozását és bejegyzését is.

Akkor indul el, amikor a felhasználó a Next Card (Következő kártya) gombra kattint, ami azt jelenti, hogy az adott kártyát fel szeretné venni a listába, és új kártyát szeretne kezdeni.

Akkor indul el, amikor a felhasználó a Save pontot választja a File menüből, ami azt jelenti, hogy az aktuális listában található összes kártyát „csomagként" szeretné menteni (például: Kvantummechanika, Hollywood-kvíz, Java-szabályok stb.).

Akkor indul el, amikor a felhasználó a New (Új) pontot választja a File menüből, ami azt jelenti, hogy új csomagot szeretne kezdeni (ezért ürítjük ki a kártyalistát és a szövegmezőket).

A SaveMenuListener hívja meg; ez végzi a tényleges fájlba írást.

```
import java.util.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
import java.io.*;

public class QuizCardBuilder {

    private JTextArea question;
    private JTextArea answer;
    private ArrayList<QuizCard> cardList;
    private JFrame frame;


    public static void main (String[] args) {
       QuizCardBuilder builder = new QuizCardBuilder();
       builder.go();
    }


    public void go() {
        // a grafikus felület felépítése
        frame = new JFrame("Quiz Card Builder");
        JPanel mainPanel = new JPanel();
        Font bigFont = new Font("sanserif", Font.BOLD, 24);
        question = new JTextArea(6,20);
        question.setLineWrap(true);
        question.setWrapStyleWord(true);
        question.setFont(bigFont);

        JScrollPane qScroller = new JScrollPane(question);
        qScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        qScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

        answer = new JTextArea(6,20);
        answer.setLineWrap(true);
        answer.setWrapStyleWord(true);
        answer.setFont(bigFont);

        JScrollPane aScroller = new JScrollPane(answer);
        aScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        aScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);


        JButton nextButton = new JButton("Next Card");

        cardList = new ArrayList<QuizCard>();

        JLabel qLabel = new JLabel("Question:");
        JLabel aLabel = new JLabel("Answer:");

        mainPanel.add(qLabel);
        mainPanel.add(qScroller);
        mainPanel.add(aLabel);
        mainPanel.add(aScroller);
        mainPanel.add(nextButton);
        nextButton.addActionListener(new NextCardListener());
        JMenuBar menuBar = new JMenuBar();
        JMenu fileMenu = new JMenu("File");
        JMenuItem newMenuItem = new JMenuItem("New");
```

Ez itt mind GUI-kód. Semmi különös nincs benne, bár a MenuBar, a Menu és a MenuItem-ek kódjára érdemes vetned egy pillantást.

```
    JMenuItem saveMenuItem = new JMenuItem("Save");
    newMenuItem.addActionListener(new NewMenuListener());

    saveMenuItem.addActionListener(new SaveMenuListener());
    fileMenu.add(newMenuItem);
    fileMenu.add(saveMenuItem);
    menuBar.add(fileMenu);
    frame.setJMenuBar(menuBar);
    frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
    frame.setSize(500,600);
    frame.setVisible(true);
}
```

Létrehozunk egy menüsávot (menuBar) és egy File menüt, majd a „new" és „save" menüelemeket (menuItem) elhelyezzük a File menüben. A menüt a menüsávhoz adjuk, majd utasítjuk a keretet, hogy ezt a menüsávot használja. A menüelemek műveleti eseményeket (ActionEvent) indítanak el.

```
public class NextCardListener implements ActionListener {
   public void actionPerformed(ActionEvent ev) {

      QuizCard card = new QuizCard(question.getText(), answer.getText());
      cardList.add(card);
      clearCard();
   }
}

public class SaveMenuListener implements ActionListener {
   public void actionPerformed(ActionEvent ev) {
      QuizCard card = new QuizCard(question.getText(), answer.getText());
      cardList.add(card);

      JFileChooser fileSave = new JFileChooser();
      fileSave.showSaveDialog(frame);
      saveFile(fileSave.getSelectedFile());
   }
}

public class NewMenuListener implements ActionListener {
   public void actionPerformed(ActionEvent ev) {
      cardList.clear();
      clearCard();
   }
}

private void clearCard() {
   question.setText("");
   answer.setText("");
   question.requestFocus();
}

private void saveFile(File file) {
  try {
      BufferedWriter writer = new BufferedWriter(new FileWriter(file));

      for(QuizCard card:cardList) {
         writer.write(card.getQuestion() + "/");
         writer.write(card.getAnswer() + "\n");
      }
     writer.close();

  } catch(IOException ex) {
      System.out.println("couldn't write the cardList out");
      ex.printStackTrace();
  }
}
```

Egy fájl-párbeszédablakot jelenít meg, és ebben a sorban várakozik, amíg a felhasználó a Save lehetőséget nem választja a párbeszédablakban. A párbeszédablakban minden navigációs, fájlkijelölési stb. műveletet a JFileChooser hajt végre. Tényleg ilyen egyszerű.

A tényleges fájlba írást végző tagfüggvény (amelyet a SaveMenuListener eseménykezelője hív meg). Az argumentuma a „File" objektum, amelyet a felhasználó ment. A File osztályt a következő oldalon vizsgáljuk meg.

Az írás hatékonyabbá tételéhez egy BufferedWriter-t láncolunk egy új FileWriter-hez. (Erről hamarosan bővebben beszélünk.)

Végigmegyünk a kártyák tömblistáján, és kiírjuk a kártyákat – soronként egyet, a kérdéseket (question) és válaszokat (answer) egy perjellel („/") elválasztva, majd új sort kezdve az „\n" újsor karakter hozzáadásával.

A java.io.File osztály

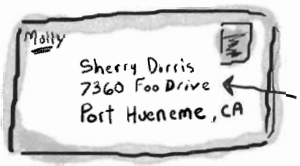
A java.io.File osztály egy fájlt jelképez vagy *ábrázol* a lemezen, de nem a fájl tényleges *tartalmát*. A File objektumokra ne úgy gondolj, mintha Maguk A Fájlok lennének – inkább olyanok, mint egy fájlhoz (vagy *könyvtárhoz*) vezető *útvonalak*. A File osztály például nem rendelkezik író-olvasó tagfüggvényekkel. A File objektumok egyik NAGYON hasznos vonása, hogy sokkal biztonságosabban ábrázolhatsz velük egy fájlt, mintha csupán egy String formátumú fájlnevet tárolnál. A legtöbb osztály, amelynek a konstruktora egy String formátumú fájlnevet vár (ilyen például a FileWriter vagy a FileInputStream), egy File objektumot is elfogad, így létrehozhatsz egy File objektumot, ellenőrizheted, hogy az elérési út érvényes-e stb., majd átadhatod ezt a File objektumot a FileWriter-nek vagy FileInputStream-nek.

A File objektumok egy-egy fájl vagy könyvtár nevét és elérési útját jelképezik a lemezen. Például: /Users/Kathy/Data/GameFile.txt A fájlban levő adatokat ugyanakkor NEM jelképezik, és nem adnak hozzájuk hozzáférést!

Néhány dolog, amit egy File objektummal csinálhatsz:

① **Egy meglevő fájlt jelképező objektummá teheted:**

```
File f = new File("MyCode.txt");
```



A cím NEM azonos magával a házzal! A File objektum olyan, mint egy cím – egy adott fájl nevét és helyét jelképezi, nem magát a fájlt.

② **Létrehozhatsz egy új könyvtárat (directory):**

```
File dir = new File("Chapter7");
dir.mkdir();
```

③ **Kiírathatod egy könyvtár tartalmát:**

```
if (dir.isDirectory()) {
      String[] dirContents = dir.list();
      for (int i = 0; i < dirContents.length; i++) {
          System.out.println(dirContents[i]);
      }
}
```

A File objektum itt a „GameFile.txt" fájlnevet jelképezi.

GameFile.txt

50,Elf,íj,kard,varázspor
200,Troll,csupasz kéz,nagy fejsze
120,Mágus,varázsigék, láthatatlanság

A File objektum NEM a fájlban levő adatokat jelképezi (és nem ad hozzájuk közvetlen hozzáférést)!

④ **Kinyerheted egy fájl vagy könyvtár abszolút elérési útját:**

```
System.out.println(dir.getAbsolutePath());
```

⑤ **Törölhetsz egy fájlt vagy könyvtárat (true-t ad vissza, ha sikerült):**

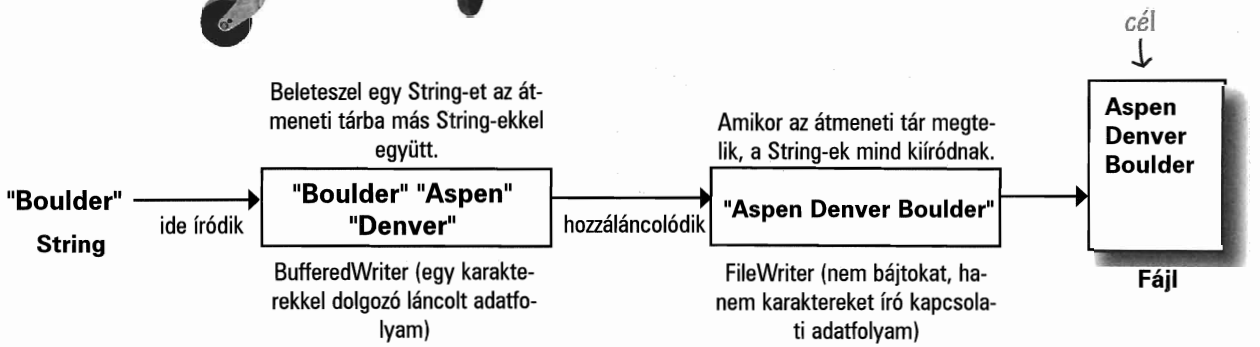
```
boolean isDeleted = f.delete();
```

Az átmeneti tárak szépsége

Ha nem lennének átmeneti tárak (buffer), az olyan lenne, mintha bevásárlókocsi nélkül kellene vásárolnod. Mindent egyesével kellene kihordanod az autóhoz, leveskonzervenként vagy wc-papírtekercsenként.



Az átmeneti tárak ideiglenes tárolóhelyet biztosítanak, amelyben elrendezheted a dolgokat, amíg a tároló (mint a bevásárlókocsi) meg nem telik. Ha átmeneti tárat használsz, sokkal kevesebbszer kell fordulnod.




```
BufferedWriter writer = new BufferedWriter(new FileWriter(aFile));
```

Figyeld meg, hogy még hivatkoznunk sem kell a FileWriter objektumra. Csak a BufferedWriter-rel törődünk, mert ez az az objektum, amelyre meghívjuk a tagfüggvényeket, és amikor bezárjuk a BufferedWriter-t, az önműködően gondoskodik a lánc többi tagjáról.

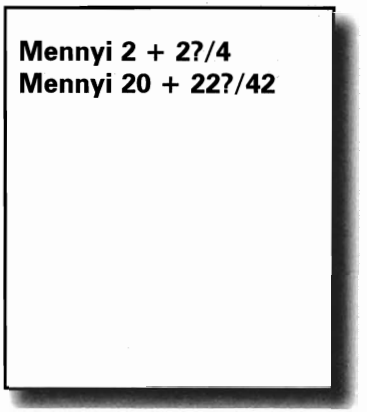
Az átmeneti tárakban az a klassz, hogy *sokkal* hatékonyabban lehet velük dolgozni, mint nélkülük. Egy fájlba önmagában a FileWriter segítségével is írhatsz a write(*valamilyenKarakterlánc*) meghívásával, de a FileWriter mindig mindent kiír, amit átadsz neki. Erre a többletterhelésre semmi szükség és nem is kívánatos, mivel minden lemezre írás Nagy Munka a memóriában levő adatokkal végzett műveletekhez képest. Ha viszont egy BufferedWriter-t láncolsz egy FileWriter-hez, a BufferedWriter tárol majd minden kiírt dolgot, amíg be nem telik, és a *FileWriter csak akkor kap tényleges utasítást a lemezen levő fájlba írásra, amikor az átmeneti tárban elfogyott a hely.*

Ha az adatok kiírására mégis az *előtt* lenne szükséged, hogy az átmeneti tár betelne, megteheted. ***Csak ki kell ürítened (flush) a tárat.*** A writer.flush() hívás azt mondja, hogy „küldd, ami a tárban van, ***most***!".

Olvasás szövegfájlból

Szöveget olvasni egy fájlból könnyű, de ezúttal egy File objektum jelképezi a fájlt, egy FileReader végzi a tényleges olvasást, és egy BufferedReader teszi azt hatékonyabbá.
Az olvasás úgy történik, hogy sorokat olvasunk be egy *while* ciklus segítségével, amelyet akkor fejezünk be, amikor egy readLine() – olvasSor – eredménye null lesz. Ez az adatok (lényegében bármi, ami nem sorosított objektum) olvasásának legelterjedtebb módja: egy while ciklussal (pontosabban egy while ciklus*teszttel*) olvasunk, és befejezzük, ha már nem maradt semmi olvasnivaló (amit onnan tudunk, hogy az olvasó tagfüggvény null eredményt ad vissza).

Fájl két sornyi szöveggel



MyText.txt

Ne feledkezz meg a betöltésről!

```
import java.io.*;

class ReadAFile {
    public static void main (String[] args) {

        try {
            File myFile = new File("MyText.txt");
            FileReader fileReader = new FileReader(myFile);

            BufferedReader reader = new BufferedReader(fileReader);

            String line = null;

            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
            reader.close();

        } catch(Exception ex) {
             ex.printStackTrace();
        }
    }
}
```

A FileReader egy kapcsolati adatfolyam karakterek számára, amely egy szövegfájlhoz kapcsolódik.

A FileReader-t a hatékonyabb olvasás kedvéért egy BufferedReaderhez láncoljuk, amely csak akkor tér vissza a fájlhoz, hogy olvasson belőle, ha az átmeneti tár kiürült (mert a program már mindent beolvasott belőle).

Létrehozunk egy String típusú változót, amelyben az egyes beolvasott sorokat tároljuk.

Ezt azt mondja, hogy „olvass be egy sornyi szöveget, és rendeld hozzá a 'line' String változóhoz. Amíg ennek a változónak az értéke nem null (mert VAN beolvasnivaló), írd ki az éppen beolvasott sort". Másként fogalmazva: „Amíg vannak beolvasandó sorok, olvasd be és írd ki őket."

QuizCardPlayer (kódvázlat)

```
public class QuizCardPlayer {

  public void go() {
    // a grafikus felület felépítése és megjelenítése
  }

  class NextCardListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // ha ez egy kérdés, megjelenítjük a választ, egyébként pedig a következő kérdést
      // beállítunk egy jelzőt, hogy kérdésről vagy válaszról van-e szó
    }
  }

  class OpenMenuListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // megjelenítünk egy fájl-párbeszédablakot, hogy a felhasználó
      // megkereshesse és kiválaszthassa a megnyitni kívánt kártyacsomagot
    }
  }

  private void loadFile(File file) {
    // fel kell építeni a kártyák tömblistáját, egy szövegfájlból beolvasva azokat
    // az OpenMenuListener hívja meg, és a fájlt soronként olvassa be, a makeCard()
    // tagfüggvényt arra utasítva, hogy minden sorból egy-egy új kártyát hozzon létre
    // (a fájl egy sora a kérdést és a választ is tartalmazza, perjellel elválasztva
  }


  private void makeCard(String lineToParse) {
    // a loadFile tagfüggvény hívja meg, és egy sort olvas be a szövegfájlból
    // a sort két részre - kérdés és válasz - bontja, és létrehoz egy új QuizCard-ot,
    // amelyet hozzáad a CardList tömblistához
  }
```

```
import java.util.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
import java.io.*;

public class QuizCardPlayer {

    private JTextArea display;
    private JTextArea answer;
    private ArrayList<QuizCard> cardList;
    private QuizCard currentCard;
    private int currentCardIndex;
    private JFrame frame;
    private JButton nextButton;
    private boolean isShowAnswer;


    public static void main (String[] args) {
       QuizCardPlayer reader = new QuizCardPlayer();
       reader.go();
    }

    public void go() {

        // a grafikus felület felépítése

        frame = new JFrame("Quiz Card Player");
        JPanel mainPanel = new JPanel();
        Font bigFont = new Font("sanserif", Font.BOLD, 24);

        display = new JTextArea(10,20);
        display.setFont(bigFont);

        display.setLineWrap(true);
        display.setEditable(false);

        JScrollPane qScroller = new JScrollPane(display);
        qScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        qScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
        nextButton = new JButton("Show Question");
        mainPanel.add(qScroller);
        mainPanel.add(nextButton);
        nextButton.addActionListener(new NextCardListener());

        JMenuBar menuBar = new JMenuBar();
        JMenu fileMenu = new JMenu("File");
        JMenuItem loadMenuItem = new JMenuItem("Load card set");
        loadMenuItem.addActionListener(new OpenMenuListener());
        fileMenu.add(loadMenuItem);
        menuBar.add(fileMenu);
        frame.setJMenuBar(menuBar);
        frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
        frame.setSize(640,500);
        frame.setVisible(true);

    }  // a go lezárása
```

Ez itt mind GUI-kód; semmi különös nincs benne.

```
public class NextCardListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
       if (isShowAnswer) {
          // megjelenítjük a választ, mert a kérdést látták
          display.setText(currentCard.getAnswer());
          nextButton.setText("Next Card");
          isShowAnswer = false;
       } else {
          // megjelenítjük a következő kérdést
          if (currentCardIndex < cardList.size()) {
             showNextCard();
          } else {
             // nincs több kártya!
             display.setText("That was last card");
             nextButton.setEnabled(false);
          }
       }
    }
}

public class OpenMenuListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
        JFileChooser fileOpen = new JFileChooser();
        fileOpen.showOpenDialog(frame);
        loadFile(fileOpen.getSelectedFile());
    }
}

private void loadFile(File file) {
    cardList = new ArrayList<QuizCard>();
    try {
        BufferedReader reader = new BufferedReader(new FileReader(file));
        String line = null;
        while ((line = reader.readLine()) != null) {
           makeCard(line);
        }
        reader.close();
    } catch(Exception ex) {
        System.out.println("couldn't read the card file");
        ex.printStackTrace();
    }
    // itt indítunk az első kártya megjelenítésével
    showNextCard();
}

private void makeCard(String lineToParse) {
    String[] result = lineToParse.split("/");
    QuizCard card = new QuizCard(result[0], result[1]);
    cardList.add(card);
    System.out.println("made a card");
}

private void showNextCard() {
     currentCard = cardList.get(currentCardIndex);
     currentCardIndex++;
     display.setText(currentCard.getQuestion());
     nextButton.setText("Show Answer");
     isShowAnswer = true;
}
} // az osztály lezárása
```

Megvizsgáljuk az isShowAnswer logikai jelzőt, hogy lássuk, a felhasználó éppen kérdést vagy választ lát, és az eredménytől függően a megfelelő dolgot csináljuk.

Megjelenítjük a fájl-párbeszédablakot, és hagyjuk, hogy a felhasználó megkeresse és kijelölje a megnyitni kívánt fájlt.

Létrehozunk egy BufferedReader-t egy új FileReader-hez láncolva, és a FileReadernek átadjuk a File objektumot, amelyet a felhasználó kiválasztott a fájlmegnyitási párbeszédablakban.
Egyenként beolvassuk a sorokat, átadva azokat a makeCard() tagfüggvénynek, amely feldolgozza, igazi kvízkártyákká alakítja, és hozzáadja őket a tömblistához.

Minden szövegsor egy-egy tanulókártyának felel meg, de még szét kell bontanunk a kérdéseket és a válaszokat. A sort a String split() tagfüggvényével vágjuk két elemre (az egyik a kérdést, a másik a választ jelképezi). A split() tagfüggvényt a következő oldalon vizsgáljuk meg.

Feldolgozás a String split() tagfüggvényével

Tegyük fel, hogy egy ilyen tanulókártyád van:



Egy ilyen fájlba mentve:

Mit kapsz, ha a kéket összekevered a sárgával?/Zöldet.
Mit kapsz, ha a pirosat összekevered a kékkel?/Lilát.

Hogyan választhatod el a kérdést a választól?

Amikor beolvasod a fájlt, a kérdés és a válasz egy sorban szerepel, és egy perjel („/") választja el őket (mivel így írtuk a fájlba a QuizCardBuilder kódjában).

A String split() tagfüggvényének segítségével részekre bonthatsz egy karakterláncot.

A split() tagfüggvény azt mondja: „add meg az elválasztót, és én a részeire bontom ezt a karakterláncot, és beleteszem a részeket egy karakterlánc-tömbbe".



1. elem **elválasztó** **2. elem**

A QuizCardPlayer alkalmazásban így néz ki egy sor, amikor beolvassuk a fájlból.

```
String toTest = "Mit kapsz, ha a kéket összekevered a sárgával?/Zöldet.";
String[] result = toTest.split("/");
for (String token:result) {
   System.out.println(token);
}
```

A split() tagfüggvény fogja a perjelet, és a mentén (ebben az esetben) két részre vágja a karakterláncot. (Megjegyzés: a split() SOKKAL többet tud annál, mint amire itt használjuk, például olyan bonyolult feldolgozási műveleteket, amelyekben szűrők, helyettesítő karakterek stb. is szerepelnek.)

Végigmegyünk a tömbön, és kiírunk minden elemet („token"-t). A példában a karakterlánc csak két elemből áll: „Mit kapsz, ha a kéket összekevered a sárgával?" és „Zöldet.".

Nincsenek

hülye kérdések

K: Megnéztem az API-t, és van vagy ötmillió osztály a java.io csomagban. Honnan a fenéből tudjam, melyiket kell használni?

V: Az I/O API a moduláris „láncolásra" épül, így a kapcsolati és a láncolt adatfolyamokat (ez utóbbiakat „szűrőfolyamoknak" is nevezik) sokféle kombinációban összekapcsolhatod, és szinte bármit elérhetsz velük, amit csak akarsz. A láncok nem csak két szintből állhatnak – több adatfolyamot is egymáshoz láncolhatsz, hogy elvégezhesd a kívánt feldolgozási műveleteket. A legtöbbször azonban ugyanazt a maroknyi osztályt fogod használni. Ha szövegfájlokat írsz, valószínűleg csak a BufferedReader és a BufferedWriter osztályra lesz szükséged (a FileReader-hez és a FileWriter-hez láncolva), ha pedig sorosított objektumokat, az ObjectOutputStream-et és az ObjectInputStream-et használhatod (a FileInputStream-hez és a FileOutputStream-hez láncolva). Más szavakkal, a Java I/O használatakor 90%-ban csak olyasmire lesz szükséged, amiről már tanultál.

K: Mi a helyzet az 1.4-es változatban megjelent új I/O-osztályokkal (nio)?

V: A java.nio csomag osztályai jelentősen növelik a műveletek sebességét, és jobban kihasználják a programodat futtató virtuális gép belső lehetőségeit. A nio egyik kulcsfontosságú új szolgáltatása, hogy közvetlenül vezérelheted az átmeneti tárakat. Egy másik új lehetőség a nem blokkoló kimenet-bemenet, ami azt jelenti, hogy az I/O-kódod nem csak ül és vár, ha nincs olvasni- vagy írnivaló. A meglevő osztályok némelyike (beleértve a FileInputStream-et és a FileOutputStream-et) a háttérben kihasználja az új lehetőségeket, a nio osztályainak használata ugyanakkor bonyolultabb, ezért hacsak *valóban* szükséged nincs az új szolgáltatásokra, érdemesebb az itt megismert egyszerűbb változatoknál maradnod. Ezenkívül, ha nem vagy elővigyázatos, a nio a sebesség *csökkenését* eredményezheti. A nem nio I/O a szokványos műveletek 90%-ában megfelel, különösen a kezdő Java-programozók számára. Mindazonáltal közelebb kerülhetsz a nio osztályaihoz, ha a FileInputStream-et használod, és a csatornáját a getChannel() tagfüggvényen keresztül éred el (ez az 1.4-es változat óta része a FileInputStream osztálynak).



Újratöltve

- Ha szövegfájlba szeretnél írni, először egy FileWriter kapcsolati adatfolyamra lesz szükséged.
- A hatékonyságot növeli, ha a FileWriter-t egy BufferedWriter-hez láncolod.
- A File objektumok egy-egy fájlt jelképeznek egy adott elérési útvonalon, de nem a fájlok tényleges tartalmát.
- Egy File objektummal könyvtárakat is létrehozhatsz, bejárhatsz és törölhetsz.
- A legtöbb adatfolyam, amely String formátumú fájlnevet vár, File objektumokat is elfogad, amelyeknek a használata sokszor biztonságosabb.
- Ha olvasni szeretnél egy szövegfájlból, először egy FileReader kapcsolati adatfolyamra lesz szükséged.
- A hatékonyságot növeli, ha a FileReader-t egy BufferedReader-hez láncolod.
- Ahhoz, hogy feldolgozhass egy szövegfájlt, meg kell győződnöd róla, hogy a fájlt olyan módon írták, hogy a különböző elemek felismerhetők benne. Az egyes részeket általában valamilyen karakterrel szokták elválasztani.
- Egy karakterláncot a String split() tagfüggvényével bonthatsz fel az elemeire. Ha egy karakterlánc egyetlen elválasztót tartalmaz, akkor két elemből („tokenből") áll (az elválasztó egyik, illetve másik oldalán). Az *elválasztó maga nem számít elemnek.*

Változatazonosító: a sorosítás Nagy Csapdája

Most már tudod, hogy a beolvasás és kiírás mennyire egyszerű a Javában, különösen ha a leggyakrabban alkalmazott kapcsolat–lánc kombinációknál maradsz. Egy kérdéssel azonban *feltétlenül* foglalkoznod kell.

A változatkövetés létfontosságú!

Ha sorosítasz egy objektumot, mindenképpen rendelkezned kell az osztállyal ahhoz, hogy visszafejthesd és felhasználhasd. Jó, ez magától értetődő. Az viszont talán kevésbé nyilvánvaló, hogy mi történik, ha időközben ***módosítod az osztályt***. Képzeld el például, hogy úgy próbálsz ismét életre kelteni egy Kutya objektumot, hogy az egyik (nem mulandó) példányváltozója közben double-ról String típusúra változott. Ez Jelentős Mértékben megsérti a Java típusbiztonsági elveit – de nem ez az egyetlen, ami ilyet tehet. Jegyezd meg az alábbiakat:

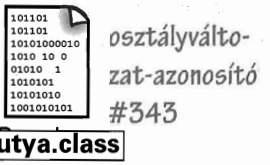
Az osztályon végrehajtott alábbi változtatások veszélyeztethetik a visszafejtést:

Egy példányváltozó törlése
Egy példányváltozó bejelentett típusának megváltoztatása
Egy nem mulandó példányváltozó mulandóra módosítása
Egy osztály feljebb vagy lejjebb helyezése az öröklési hierarchiában
Egy (az objektumgráf tetszőleges részén található) sorosítható osztály nem sorosíthatóvá tétele (az „implements Serializable" kitétel eltávolításával az osztálydeklarációból)
Egy példányváltozó statikussá tétele

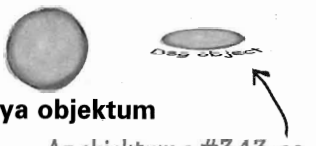
Az osztályon végrehajtott alábbi változtatások általában veszélytelenek:

Új példányváltozók hozzáadása az osztályhoz (a meglevő objektumoknak azok a példányváltozói, amelyek a sorosításkor még nem léteztek, visszafejtéskor az alapértelmezett értékeket kapják)
Osztályok hozzáadása az öröklési fához
Osztályok eltávolítása az öröklési fából
Egy példányváltozó hozzáférési szintjének módosítása nincs hatással a visszafejtésnek arra a képességére, hogy értéket rendelhet a változóhoz
Példányváltozó mulandóról nem mulandóvá tétele (a korábban sorosított objektumok egyszerűen alapértelmezett értékeket adnak a korábban mulandó változóknak)

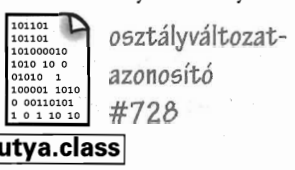
1 Írsz egy Kutya osztályt



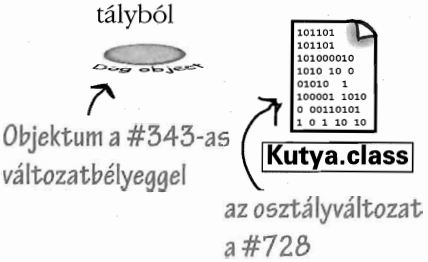
2 Sorosítasz egy Kutya objektumot ebből az osztályból



3 Módosítod a Kutya osztályt



4 Visszafejtesz egy Kutya objektumot a módosított osztályból



5 A sorosítás meghiúsul!!!
A JVM azt mondja, hogy egy „öreg Kutyát nem taníthatsz új kódra".

A serialVersionUID használata

Minden alkalommal, amikor sorosítasz egy objektumot, az objektum (és a gráfjában található összes objektum) kap egy „bélyeget" az objektum osztályának a változatazonosító számával. Ezt az azonosítót serialVersionUID-nek hívják, és a kiszámítása az osztályszerkezetet leíró információk alapján történik. Ha az osztály a sorosítás óta megváltozott, amikor az objektumot visszafejted, az osztálynak különböző serialVersionUID-je lehet, ami meghiúsítja a visszafejtést, de a problémára szerencsére van megoldás.

Ha úgy gondolod, hogy VAN rá esély, hogy az osztály *továbbfejlődik*, tegyél egy változatazonosító számot az osztályba.

Amikor a Java megpróbál visszafejteni egy objektumot, összehasonlítja a sorosított objektum serialVersionUID-jét annak az osztálynak a változatazonosítójával, amelyet a JVM a visszafejtéshez használ. Ha egy Dog-példányt például a 23 azonosítóval sorosítottak (a valóságban a serialVersionUID-k sokkal hosszabbak), a JVM a Dog objektum visszafejtésekor először összeveti a Dog objektum serialVersionUID-jét a Dog osztály serialVersionUID-jével. Ha a két szám nem egyezik, a JVM feltételezi, hogy az osztály nem összeegyeztethető a korábban sorosított objektummal, ezért a visszafejtéskor kivételt kapsz.

A megoldás tehát az, hogy az osztályban elhelyezel egy serialVersionUID-t, így az osztály továbbfejlődése során a változatazonosító ugyanaz marad, és a JVM elfogadja az osztályt az adott sorosított objektumhoz, akkor is, ha az osztály egyébként megváltozott.

Ez persze *csak akkor* működik, ha körültekintő vagy az osztály változásaival kapcsolatban. Más szavakkal, a *tiéd* minden felelősség, ha egy régebbi objektumot egy újabb osztállyal keltesz életre.

Egy osztály változatazonosító számát a Java-fejlesztőcsomagban található serialver segédprogram segítségével kaphatod meg:

```
File Edit Window Help serialKiller
% serialver Dog
Dog: static final long
serialVersionUID = -
5849794470654667210L;
```

Ha úgy gondolod, hogy az osztályod változhat, miután valaki objektumokat sorosított belőle...

1. A serialver parancssori eszköz segítségével szerezd meg az osztály változatazonosító számát

```
File Edit Window Help serialKiller
% serialver Dog
Dog: static final long
serialVersionUID = -
5849794470654667210L;
```

2. A kimenetet illeszd be az osztályba

```
public class Dog {

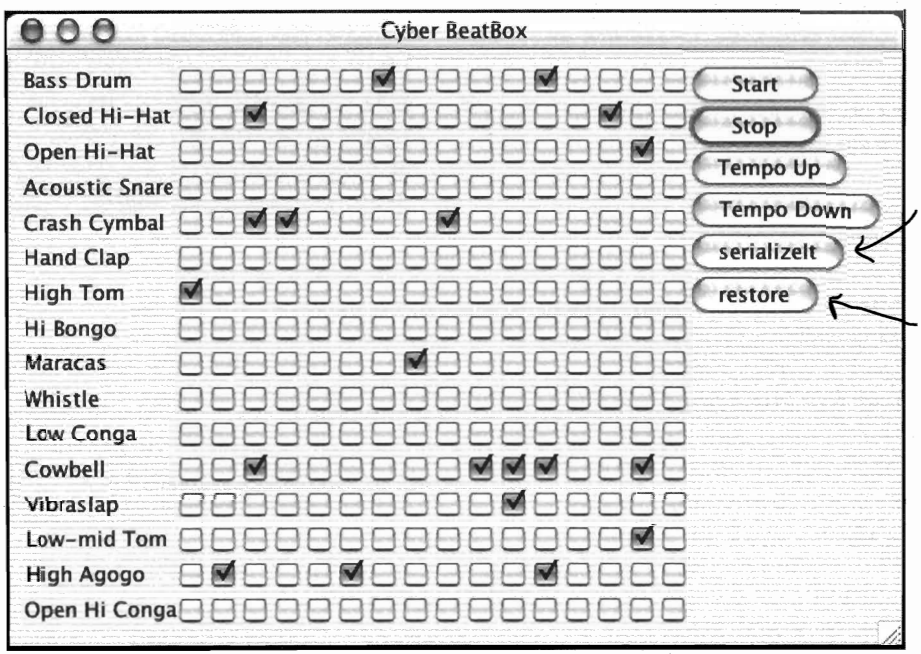
  static final long serialVersionUID =
                  -6849794470754667710L;

  private String name;
  private int size;

    // tagfüggvénykód
}
```

3. Ne feledd, hogy ha módosítod az osztályt, a kódban a te felelősséged kezelni az osztályon végrehajtott változtatások következményeit. Például gondoskodnod kell róla, hogy az új Dog osztályod meg tudjon birkózni egy régi Dog visszafejtésével, amelyben a sorosítás *után* az osztályhoz adott példányváltozók alapértelmezett értékeket kapnak.

Kódkonyha



Amikor a „serializelt" gombra kattintasz, a program menti az aktuális mintát.

A „restore" visszatölti a mentett mintát, és alaphelyzetbe állítja a jelölőnégyzeteket.

Tedd képessé a BeatBox-ot a kedvenc dobmintád mentésére és visszaállítására!

BeatBox-minta mentése

Emlékezz vissza, hogy a BeatBox-ban egy dobminta valójában csak egy csomó jelölőnégyzet. Amikor egy utasítássorozatot le kell játszani, a kód végigmegy a jelölőnégyzeteken, hogy kiderítse, mely dobhangoknak mikor kell megszólalniuk a 16 ütem alatt. Egy minta mentéséhez tehát csak a jelölőnégyzetek állapotát kell mentened. Létrehozhatsz egy egyszerű logikai tömböt, amely a 256 jelölőnégyzethez tartozó állapotokat tárolja. A tömbobjektumok egészen addig sorosíthatók, amíg a *tömbben* levő dolgok is azok, így egy boolean értékekből álló tömb mentése nem jelent gondot.
Ha vissza akarsz tölteni egy mintát, csak be kell olvasnod (visszafejtened) ezt az egyetlen logikai tömbobjektumot, és visszaállítani a jelölőnégyzeteket. A kód nagy részét már láttad abban a Kódkonyhában, ahol a BeatBox grafikus felületét építettük fel, ezért ebben a fejezetben csak a mentés és visszaállítás kódját nézzük meg.
Ez a Kódkonyha egyben felkészít a következő fejezetre, ahol a mintákat nem *fájlba* írjuk majd, hanem a *hálózaton* át elküldjük a kiszolgálónak, és nem fájlból töltjük *be*, hanem a *kiszolgálótól* kérjük el őket, amikor csak valamelyik felhasználó feltölt egyet oda.

Minta sorosítása

Ez egy belső osztály a BeatBox-kódon belül.

```
public class MySendListener implements ActionListener {

    public void actionPerformed(ActionEvent a) {
        boolean[] checkboxState = new boolean[256];

        for (int i = 0; i < 256; i++) {

            JCheckBox check = (JCheckBox) checkboxList.get(i);
            if (check.isSelected()) {
               checkboxState[i] = true;
            }
        }

        try {
           FileOutputStream fileStream = new FileOutputStream(new File("Checkbox.ser"));
           ObjectOutputStream os = new ObjectOutputStream(fileStream);
           os.writeObject(checkboxState);
        } catch(Exception ex) {
            ex.printStackTrace();
        }

    } // tagfüggvény lezárása
} // belső osztály lezárása
```

Ez akkor fut le, amikor a felhasználó a gombra kattint, és elindul az ActionEvent.

Létrehozunk egy logikai tömböt az egyes jelölőnégyzetek állapotának tárolására.

Végigmegyünk a checkboxList-en (a jelölőnégyzetek tömblistáján), kiolvassuk az egyes jelölőnégyzetek állapotát, és hozzáadjuk azokat a logikai tömbhöz.

Ez a rész gyerekjáték: csak ki kell írni (sorosítani kell) az egyetlen logikai tömböt.

BeatBox-minta visszaállítása

Ez lényegében ugyanaz, csak fordítva... Kiolvasod a logikai tömböt, és a segítségével visszaállítod a grafikus felület jelölőnégyzeteinek állapotát. Mindez akkor történik, amikor a felhasználó a „restore" gombra kattint.

Minta visszaállítása

Ez egy másik belső osztály a BeatBox osztályon belül.

```
public class MyReadInListener implements ActionListener {

    public void actionPerformed(ActionEvent a) {
       boolean[] checkboxState = null;
       try {
           FileInputStream fileIn = new FileInputStream(new File("Checkbox.ser"));
           ObjectInputStream is = new ObjectInputStream(fileIn);
           checkboxState = (boolean[]) is.readObject();
       } catch(Exception ex) {ex.printStackTrace();}

       for (int i = 0; i < 256; i++) {
          JCheckBox check = (JCheckBox) checkboxList.get(i);
          if (checkboxState[i]) {
             check.setSelected(true);
          } else {
             check.setSelected(false);
           }
       }

       sequencer.stop();
       buildTrackAndStart();

    } // tagfüggvény lezárása
} // belső osztály lezárása
```

Beolvassuk a fájlban található egyetlen objektumot (a logikai tömböt), és visszaalakítjuk boolean tömbbé (emlékezz vissza, hogy a readObject() Object típusú hivatkozást ad vissza).

Most visszaállítjuk a JCheckBox objektumok tömblistájában (checkboxList) levő jelölőnégyzetek állapotát.

Leállítjuk az éppen lejátszott mintát, és újraépítjük az utasítássorozatot a tömblistában levő jelölőnégyzetek új állapota alapján.



Hegyezd ki a ceruzád!

Ennek a változatnak van egy komoly hiányossága. Amikor a „serializeIt" gombra kattintasz, automatikusan sorosít egy „Checkbox.ser" nevű fájlba (ami létrejön, ha még nem létezik), de minden mentéskor felülírod ezt a fájlt.
Javíts a mentési és visszaállítási szolgáltatáson egy JFileChooser beépítésével, hogy tetszőleges számú mintát nevezhess el és menthess, és bármelyik korábban mentett mintafájlt betölthesd, illetve visszaállíthasd!



Hegyezd ki a ceruzád!

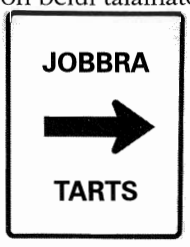
Menthetők?

Az alábbiak közül szerinted melyeket lehet vagy célszerű sorosítani? Ha nem, miért nem? Nincs értelme? Biztonsági kockázatot jelent? Csak a JVM adott végrehajtásakor működik? Anélkül válaszolj, hogy belenéznél az API-ba!

Objektum típusa	Sorosítható?	Ha nem, miért nem?
Object	Igen / Nem	_____
String	Igen / Nem	_____
File	Igen / Nem	_____
Date	Igen / Nem	_____
OutputStream	Igen / Nem	_____
JFrame	Igen / Nem	_____
Integer	Igen / Nem	_____
System	Igen / Nem	_____

Melyik érvényes?

Karikázd be azokat a kódtöredékeket, amelyek lefordíthatók (feltételezve, hogy egy érvényes osztályon belül találhatók)!




```
FileReader fileReader = new FileReader();
BufferedReader reader = new BufferedReader(fileReader);
```

```
FileOutputStream f = new FileOutputStream(new File("Foo.ser"));
ObjectOutputStream os = new ObjectOutputStream(f);
```

```
BufferedReader reader = new BufferedReader(new FileReader(file));
String line = null;
while ((line = reader.readLine()) != null) {
     makeCard(line);
}
```

```
ObjectInputStream is = new ObjectInputStream(new FileOutputStream("Game.ser"));
GameCharacter oneAgain = (GameCharacter) is.readObject();
```



Ez a fejezet feltárta előtted a Java I/O csodálatos világát. A feladatod az, hogy eldöntsd, hogy a bemenettel-kimenettel kapcsolatos alábbi állítások közül melyik igaz, és melyik hamis.

Igaz vagy hamis?

1. A sorosításra csak akkor van szükség, ha nem Java nyelvű programok számára kell adatokat mentened.
2. Az objektumok állapota csak sorosítással menthető.
3. Az ObjectOutputStream osztály sorosított objektumok mentésére használatos.
4. A láncolt adatfolyamok önmagukban és kapcsolati adatfolyamokkal összekapcsolva is használhatók.
5. Egyetlen writeObject() hívással több objektum is menthető.
6. Alapértelmezés szerint minden osztály sorosítható.
7. A példányváltozókat a transient módosító segítségével lehet sorosíthatóvá tenni.
8. Ha egy ősosztály nem sorosítható, akkor az alosztályai sem azok.
9. Az objektumok beolvasása visszafejtéskor „utolsó be, első ki" sorrendben történik.
10. Amikor egy objektumot visszafejtesz, a konstruktora nem fut le.
11. A sorosítás és a szövegfájlba mentés egyaránt kiválthat kivételeket.
12. Egy BufferedWriter hozzáláncolható egy FileWriter-hez.
13. A File objektumok csak fájlokat jelképezhetnek, könyvtárakat nem.
14. Egy átmeneti tárat nem kényszeríthetsz arra, hogy elküldje a benne levő adatokat, mielőtt megtelne.
15. A fájlolvasók és a fájlírók egyaránt használhatnak átmeneti tárat.
16. A String osztály split() tagfüggvénye az elválasztókat is belefoglalja elemként (token) az eredménytömbbe.
17. Egy osztályon végrehajtott *bármilyen* változtatás működésképtelenné teszi az adott osztályból korábban sorosított objektumokat.





Kódmágnesek

Ez a feladat nehezebb, ezért Gyakorlatból teljes értékű Fejtörővé léptettük elő. Állítsd össze a kódrészletekből azt a működőképes Java-programot, amely a lap alján látható kimenetet eredményezi. (Lehet, hogy nem lesz szükséged valamennyi kódmágnesre, de egy mágnest egynél többször is felhasználhatsz.)

```
class DungeonGame implements Serializable {
try {
FileOutputStream fos = new
FileOutputStream("dg.ser");
short getZ() {
return z;
e.printStackTrace();
oos.close();
ObjectInputStream ois = new
ObjectInputStream(fis);
int getX() {
return x;
System.out.println(d.getX()+d.getY()+d.getZ());
FileInputStream fis = new
FileInputStream("dg.ser");
public int x = 3;
transient long y = 4;
private short z = 5;
long getY() {
return y;
class DungeonTest {
ois.close();
import java.io.*;
fos.writeObject(d);
} catch (Exception e) {
d = (DungeonGame) ois.readObject();
ObjectOutputStream oos = new
ObjectOutputStream(fos);
oos.writeObject(d);
public static void main(String [] args) {
DungeonGame d = new DungeonGame();
```

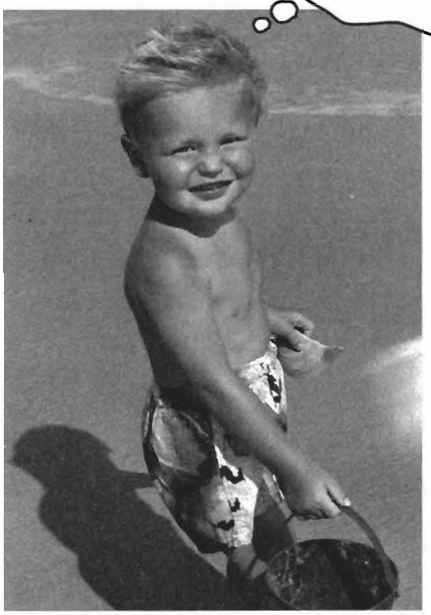
```
File Edit Window Help Torture
% java DungeonTest
12
8
```



Gyakorlat – megoldás

1. A sorosításra csak akkor van szükség, ha nem Java nyelvű programok számára kell adatokat mentened. **Hamis**
2. Az objektumok állapota csak sorosítással menthető. **Hamis**
3. Az ObjectOutputStream osztály sorosított objektumok mentésére használatos. **Igaz**
4. A láncolt adatfolyamok önmagukban és kapcsolati adatfolyamokkal összekapcsolva is használhatók. **Hamis**
5. Egyetlen writeObject() hívással több objektum is menthető. **Igaz**
6. Alapértelmezés szerint minden osztály sorosítható. **Hamis**
7. A példányváltozókat a transient módosító segítségével lehet sorosíthatóvá tenni. **Hamis**
8. Ha egy ősosztály nem sorosítható, akkor az alosztályai sem azok. **Hamis**
9. Az objektumok beolvasása visszafejtéskor „utolsó be, első ki" sorrendben történik. **Hamis**
10. Amikor egy objektumot visszafejtesz, a konstruktora nem fut le. **Igaz**
11. A sorosítás és a szövegfájlba mentés egyaránt kiválthat kivételeket. **Igaz**
12. Egy BufferedWriter hozzáláncolható egy FileWriter-hez. **Igaz**
13. A File objektumok csak fájlokat jelképezhetnek, könyvtárakat nem. **Hamis**
14. Egy átmeneti tárat nem kényszeríthetsz arra, hogy elküldje a benne levő adatokat, mielőtt megtelne. **Hamis**
15. A fájlolvasók és a fájlírók egyaránt használhatnak átmeneti tárat. **Igaz**
16. A String osztály split() tagfüggvénye az elválasztókat is belefoglalja elemként (token) az eredménytömbbe. **Hamis**
17. Egy osztályon végrehajtott *bármilyen* változtatás működésképtelenné teszi az adott osztályból korábban sorosított objektumokat. **Hamis**






```
import java.io.*;

class DungeonGame implements Serializable {
  public int x = 3;
  transient long y = 4;
  private short z = 5;
  int getX() {
    return x;
  }
  long getY() {
    return y;
  }
  short getZ() {
    return z;
  }
}

class DungeonTest {
  public static void main(String [] args) {
    DungeonGame d = new DungeonGame();
    System.out.println(d.getX() + d.getY() + d.getZ());
    try {
      FileOutputStream fos = new FileOutputStream("dg.ser");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(d);
      oos.close();
      FileInputStream fis = new FileInputStream("dg.ser");
      ObjectInputStream ois = new ObjectInputStream(fis);
      d = (DungeonGame) ois.readObject();
      ois.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
    System.out.println(d.getX() + d.getY() + d.getZ());
  }
}
```

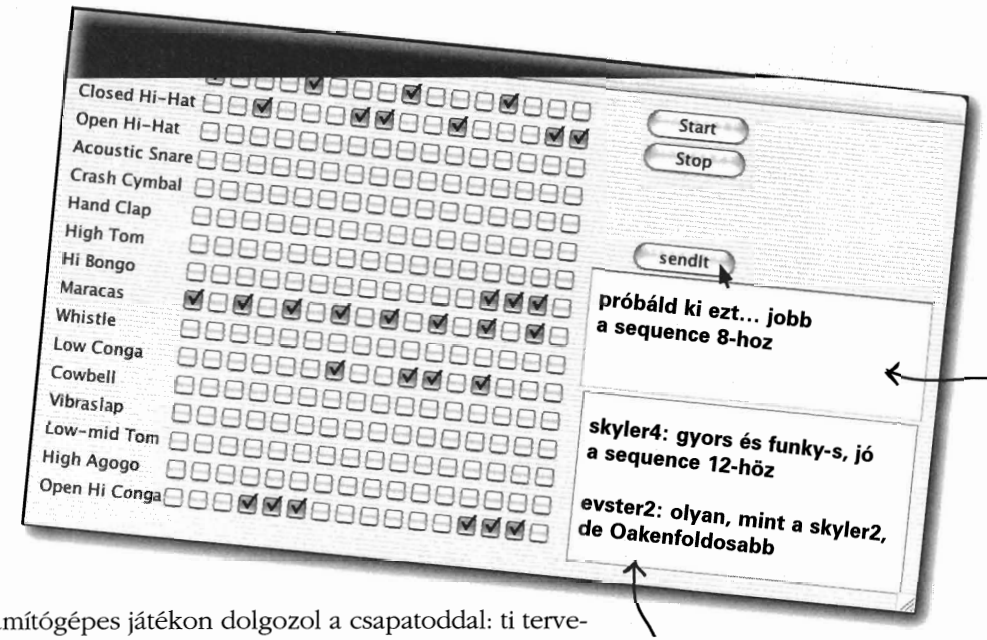
```
File  Edit  Window  Help  Escape
% java DungeonTest
12
8
```

Kapcsolatfelvétel



Vedd fel a kapcsolatot a külvilággal! A Java-programjaid kinyújthatják a kezüket, és megérinthetnek egy másik számítógépen található programot. A dolog egyszerű, mert a hálózatkezelés minden alacsonyszintű részletéről a java.net könyvtár osztályai gondoskodnak. A Java egyik nagy előnye, hogy az adatok küldése és fogadása egy hálózaton keresztül is csak egyszerű bemenet-kimenet, csak éppen kissé különböző kapcsolati adatfolyammal a lánc végén. Ha van egy BufferedReader-ed, akkor *olvashatsz*. A BufferedReader-t egyáltalán nem érdekli, hogy az adatok egy fájlból származnak vagy egy Ethernet-kábelen át érkeztek. Ebben a fejezetben csatolók (socket) segítségével kapcsolódsz majd a külvilághoz. Létrehozol *ügyfél*csatolókat (client socket) és *kiszolgálói* csatolókat (server socket), *ügyfeleket* és *kiszolgálókat*, és ráveszed őket arra, hogy társalogjanak egymással. Mielőtt a fejezet végére érnél, lesz egy teljesen működőképes, többszálú (multithreaded) csevegő-ügyfélprogramod. Igen, tényleg azt mondtuk, hogy *többszálú*: végre *megtudod* a titkát annak, hogy miként beszélhetsz Bobbal, miközben Suzy-t hallgatod.

Valósidejű BeatBox-csevegés

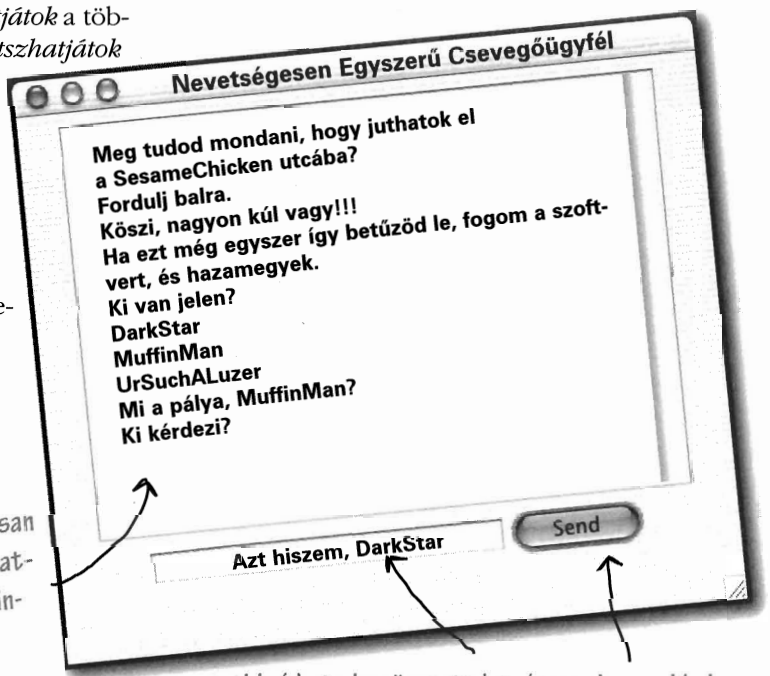


Írj be egy üzenetet, és nyomd meg a sendIt gombot az üzenet ÉS az aktuális dobminta elküldéséhez.

Ha rákattintasz egy fogadott üzenetre, betöltöd a vele kapott mintát.

Egy számítógépes játékon dolgozol a csapatoddal: ti tervezitek a játék minden részének a hangját. A BeatBox „csevegő" változatának segítségével együttműködhettek – az üzenetekhez mellékelhettek egy dobmintát, és azt minden résztvevő megkapja. Tehát nem csak *olvashatjátok* a többiek üzeneteit, hanem be is tölthetitek és *lejátszhatjátok* a dobmintákat, egyszerűen az üzenetre kattintva a bejövő üzeneteket megjelenítő területen.

Ebben a fejezetben megtanulod, mi kell egy ilyen csevegőügyfél felépítéséhez, sőt még a csevegő*kiszolgálók* készítéséről is megtudsz ezt-azt. A teljes Beat Box Chat kódját a Kódkonyha részre tartogatjuk, de ne aggódj: a fejezetben *el fogsz* készíteni egy Nevetségesen Egyszerű Csevegőügyfelet és egy Nagyon Egyszerű Csevegőkiszolgálót, amelyekkel szöveges üzeneteket küldhetsz és fogadhatsz.

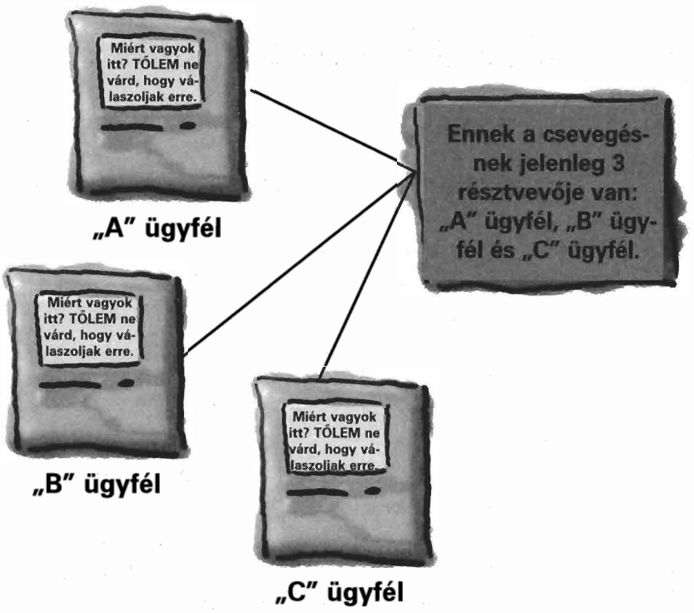


Magasröptű, intellektuálisan ösztönző csevegést folytathattok. Az üzeneteket minden résztvevő megkapja.

Ide írhatod az üzenetedet, és ezzel a gombbal küldheted el a kiszolgálónak.

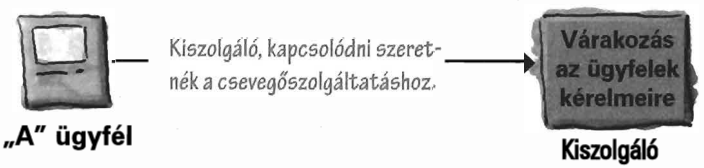
A csevegőprogram áttekintése

Az Ügyfélnek ismernie kell a Kiszolgálót.
A Kiszolgálónak ismernie kell MINDEN Ügyfelet.

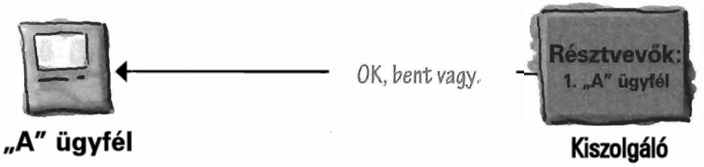


Hogyan működik?

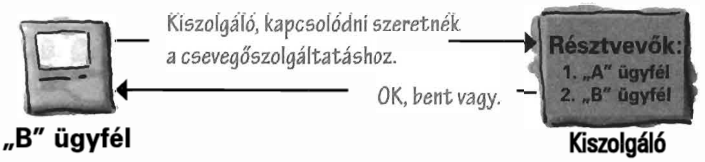
1. Az ügyfél (client) kapcsolódik a kiszolgálóhoz (server)



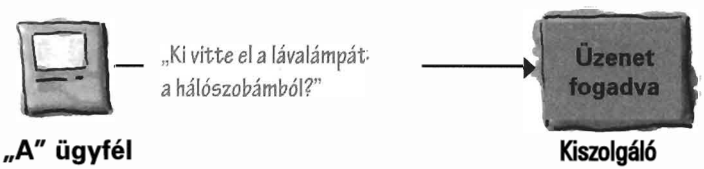
2. A kiszolgáló létrehoz egy kapcsolatot, és hozzáadja az ügyfelet a résztvevők listájához



3. Egy másik ügyfél is kapcsolódik



4. Az „A" ügyfél üzenetet küld a csevegőszolgáltatásnak



5. A kiszolgáló kézbesíti az üzenetet MINDEN résztvevőnek (beleértve a küldőt is)

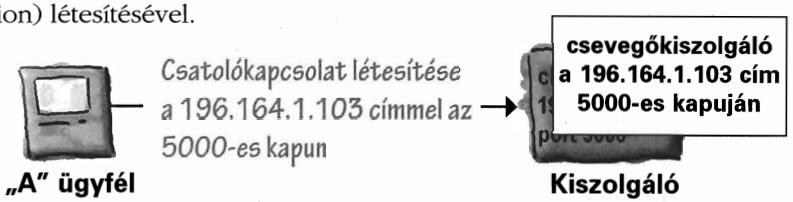


Kapcsolódás, küldés és fogadás

A következő három dolgot kell megtanulnod ahhoz, hogy működésre bírd az ügyfelet:
1) Hogyan teremtheted meg a kezdeti **kapcsolatot** az ügyfél és a kiszolgáló között?
2) Hogyan **küldhetsz** üzeneteket a kiszolgáló*nak*?
3) Hogyan **fogadhatsz** üzeneteket a kiszolgáló*tól*?
Ahhoz, hogy ezek a dolgok működjenek, egy csomó alacsonyszintű műveletet kell végrehajtani, de szerencséd van, mert a Java API hálózatkezelési csomagjával (java.net) gyerekjáték az egész. Grafikus felületi kóddal sokkal többel fogsz találkozni, mint hálózatkezelő vagy I/O-kóddal.
És ez még nem minden.
Az egyszerű csevegőügyfélen belül leselkedik egy probléma, amivel eddig még nem találkoztál ebben a könyvben: két dolgot csinálni egyszerre. A kapcsolatfelvétel egyszeri művelet (ami vagy sikerül, vagy nem), de utána a csevegés résztvevői *üzeneteket szeretnének küldeni* és **egyidejűleg** *üzeneteket akarnak fogadni* a többi résztvevőtől (a kiszolgálón keresztül). Hm... ez némi töprengést igényel, de pár oldal, és a kezedben lesz a megoldás.

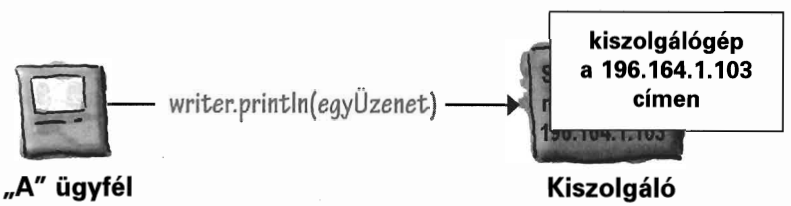
1 Kapcsolódás

Az ügyfél kapcsolódik a kiszolgálóhoz egy **csatolókapcsolat** (socket connection) létesítésével.



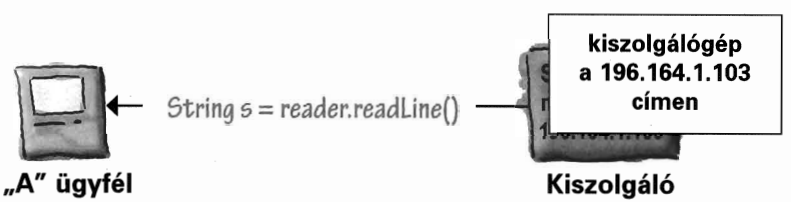
2 Küldés

Az ügyfél üzenetet **küld** a kiszolgálónak.



3 Fogadás

Az ügyfél üzenetet **kap** a kiszolgálótól.



Hálózati csatolókapcsolat létesítése

Ahhoz, hogy egy másik számítógéphez csatlakozz, egy csatolókapcsolatra van szükséged. A csatoló (Socket; java.net.Socket osztály) egy olyan objektum, amely egy hálózati kapcsolatot jelképez két számítógép között. És mi a kapcsolat? *Összeköttetés* **két, különböző számítógépen futó program** között. A legfontosabb azonban az, hogy ez a két program tudja, hogyan *kommunikálhat* egymással, vagyis hogy miként küldhetnek egymásnak *biteket*.

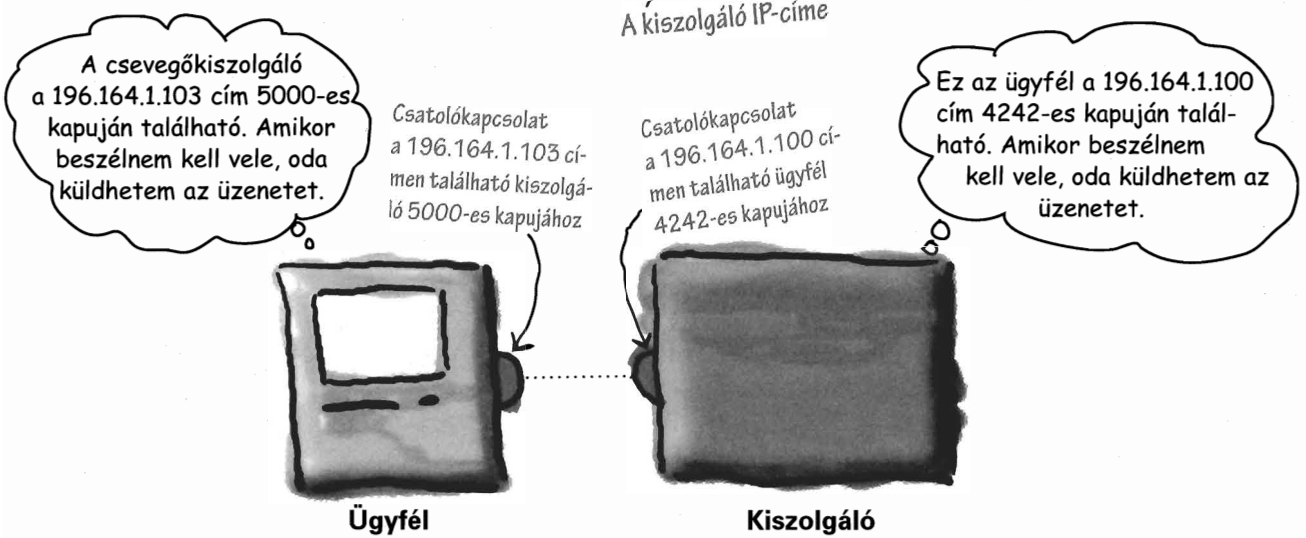
Szerencsére az alacsonyszintű részletekkel nem kell törődnöd, mert ezeket a műveleteket a „hálózati verem" sokkal lejjebb levő részei kezelik. Nem baj, ha nem tudod, mi az a „hálózati verem" – csupán egy nézete azoknak a rétegeknek, amelyeken az információknak (biteknek) át kell haladniuk, hogy egy operációs rendszeren működő JVM-ben futó Java-programtól egy fizikai hardvereszközön (mondjuk Ethernet-kábelen) keresztül eljussanak egy másik számítógépre. *Valakinek* persze ezekkel a piszkos részletekkel is foglalkoznia kell – de nem neked. Ez a valaki az adott operációs rendszer megfelelő programjának, illetve a Java hálózatkezelési API-jának együttese. Neked csak a magasszintű – mondjuk inkább úgy, hogy *nagyon* magas szintű – részletekről kell gondoskodnod, ami megdöbbentően egyszerű. Készen állsz?

Egy csatolókapcsolat létesítéséhez két dolgot kell tudnod a kiszolgálóról: kicsoda és melyik kapunk fut.
Más szavakkal:
az IP-címét és a TCP-kapuszámát.

TCP-kapuszám

```
Socket chatSocket = new Socket("196.164.1.103", 5000);
```

A kiszolgáló IP-címe



A csatolókapcsolat azt jelenti, hogy a két számítógép információkkal rendelkezik egymásról, például tudják egymás hálózati helyét (IP-címét) és TCP-kapuját.

A TCP-kapu csak egy szám: egy 16 bites szám, amely egy adott programot azonosít a kiszolgálón.

Az internetes (HTTP-) webkiszolgálód a 80-as kapun (port) fut. Ez a szabvány. A Telnet-kiszolgálóé a 23-as kapu, az FTP-é a 20-as, a POP3 levélkiszolgálóé a 110-es, az SMTP-é a 25-ös, az időkiszolgálóé (Time) pedig a 37-es. A kapuszámokra amolyan egyedi azonosítókként gondolhatsz, amelyek a kiszolgáló adott programjaihoz vezető logikai kapcsolatokat jelképeznek. Ennyi az egész. A számítógép házát hiába forgatod, nem fogsz TCP-kaput találni rajta. Először is, egy kiszolgálón 65 536 darab (0 – 65 535) található belőlük – világos, hogy nem tartozhat mindegyikhez fizikai csatlakozó. A TCP-kapuk csupán számok, amelyek egy-egy alkalmazást jelképeznek.

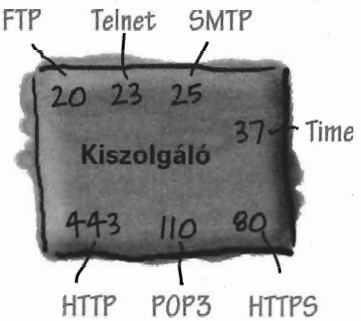
Kapuszámok nélkül a kiszolgáló nem tudná kideríteni, hogy az ügyfél melyik alkalmazáshoz szeretne kapcsolódni. Ezen kívül, mivel minden alkalmazás saját, egyedi protokollal rendelkezhet, komoly gondot okozna, ha nem lennének ezek az azonosítók. Mi történne például, ha a webböngésződ a POP3-levélkiszolgálón találná magát a HTTP-kiszolgáló helyett? A levélkiszolgáló nem tudja, hogyan kell értelmezni egy HTTP-kérelmet – de még ha tudná is, semmit sem tud arról, hogy mit kell vele kezdeni.

Amikor kiszolgálóprogramot írsz, bele kell foglalnod azt a kódot, ami elárulja a programnak, hogy melyik kapuszámon szeretnéd futtatni. (Még ebben a fejezetben megtudod, hogyan kell ezt csinálni a Javában.) A fejezetünkben elkészíteni kívánt csevegőprogramhoz az 5000-es kaput választottuk. Hogy miért éppen azt? Csak. És persze azért, mert megfelelt annak a követelménynek, hogy egy 1024 és 65 535 közötti szám legyen – a 0 és 1023 közötti kapuszámok ugyanis az olyan jól ismert szolgáltatásoknak vannak fenntartva, mint amilyeneket feljebb említettünk.

Ha egy céges hálózat számára készítesz szolgáltatásokat (kiszolgálói programokat), egyeztetned kell a rendszergazdával, hogy mely kapuk foglaltak. A rendszergazda közölheti például, hogy a 3000-es alatti kapuszámokat egyáltalán nem használhatod.

Ha kedves az életed, semmiképpen ne jelölj ki kapuszámokat hasraütésszerűen – hacsak nem a saját *otthoni* hálózatodról van szó (ebben az esetben azonban a *srácaiddal* kell egyeztetned).

Gyakran használt kiszolgálói alkalmazások ismert TCP-kapuszámai



Egy kiszolgálón 65 536 különböző kiszolgálói alkalmazás futhat – kapunként egy.

A 0 és 1023 közötti TCP-kapuszámok az ismert szolgáltatások számára vannak fenntartva, ezért ne használd őket a saját kiszolgálói programjaidhoz!*

Az elkészíteni kívánt csevegőkiszolgálónk az 5000-es kaput használja. A kapuszámot véletlenszerűen választottuk ki az 1024 és 65 535 közötti számok halmazából.

*Pontosabban, *lehet*, hogy használhatod valamelyiküket, de a helyi rendszergazda valószínűleg meg fog ölni.

Nincsenek

hülye kérdések

K: Honnan tudhatom a kapuszámát annak a kiszolgálói programnak, amellyel társalogni szeretnék?

V: Ez attól függ, hogy egy jól ismert szolgáltatásról van-e szó. Ha egy olyan ismert szolgáltatáshoz próbálsz kapcsolódni, mint amilyeneket az előző oldalon felsoroltunk (HTTP, SMTP, FTP stb.), a kapuszámot megnézheted az Interneten (Google: „Well-known TCP Port"), vagy megkérdezheted a legközelebbi barátságos rendszergazdától. Ha azonban a program nem az ismert szolgáltatások közé tartozik, a kapuszámot a működtetőjétől kell megtudakolnod. Kérdezd meg tőle. Általában persze ha valaki olyan hálózati szolgáltatást készít, amelyhez azt szeretné, hogy mások ügyfélprogramokat írjanak, közzéteszi a szolgáltatás IP-címét, kapuszámát és protokollját. Például ha egy go-játékkiszolgálóhoz szeretnél ügyfelet írni, ellátogathatsz valamelyik go-kiszolgáló webhelyére, ahol megtalálod az adott kiszolgálóval működő ügyfelek elkészítéséhez szükséges információkat.

K: Egy kapun egyszerre több program is futhat? Másként fogalmazva, lehet két alkalmazásnak ugyanazon a kiszolgálón ugyanaz a kapuszáma?

V: Nem! Ha egy programot egy már használatban levő kapuhoz próbálsz kötni, BindException kivételt kapsz. Egy program kapuhoz *kötése* (bind) csupán annyit jelent, hogy elindítod a kiszolgálói alkalmazást, és utasítod, hogy egy adott kapun fusson. A fejezet kiszolgálókkal foglalkozó részében erről is bővebben beszélünk majd.

Az IP-cím a bevásárlóközpont.



Az IP-cím olyan, mint egy adott bevásárlóközpont – mondjuk „Java Pláza" – meghatározása.

A kapuszám a bevásárlóközpont konkrét üzlete.



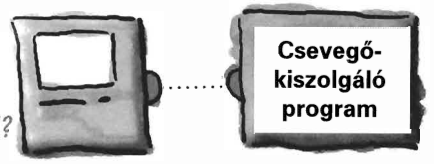
A kapuszám olyan, mint egy konkrét üzlet – mondjuk „Bob CD-boltja" – megnevezése.



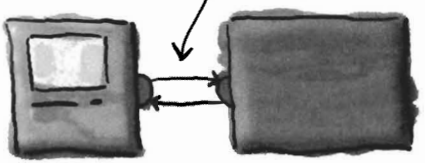
Agygyúró

OK, van egy csatolókapcsolatod. Az ügyfél és a kiszolgáló ismerik egymás IP-címét és TCP-kapuszámát. Hogyan tovább? Hogyan kommunikálsz ezen a kapcsolaton keresztül? Más szavakkal, hogyan mozgatod a biteket a két fél között? Gondold végig, hogy milyen üzeneteket kell küldenie és fogadnia a csevegőügyfélnek!

Pontosan hogyan társalognak ezek ketten egymással?



Adatokat egy BufferedReader segítségével olvashatsz egy csatolóról



A csatolókapcsolatokon keresztül adatfolyamok segítségével kommunikálhatsz – ugyanazokról a régi jó I/O-adatfolyamokról van szó, amelyeket az előző fejezetben már használtál. A Java egyik legklasszabb vonása, hogy a bemeneti-kimeneti műveletek többségét nem érdekli, hogy a magasszintű láncolt adatfolyamod valójában mihez kapcsolódik. Vagyis egy BufferedReader-t ugyanúgy használhatsz, mint amikor fájlba írtál – a különbség csak annyi lesz, hogy a háttérben működő kapcsolati adatfolyam nem egy *fájlhoz*, hanem egy *csatolóhoz* fog kapcsolódni.

1 Létrehozol egy csatolókapcsolatot (Socket) a kiszolgálóval

```
Socket chatSocket = new Socket("127.0.0.1", 5000);
```

A kapuszám, amit onnan tudsz, hogy MEGMONDTUK neked, hogy az 5000 a csevegőkiszolgálónk kapuszáma.

A 127.0.0.1 a „helyi állomás" (localhost) IP-címe – ez az a hely, ahol ez a kód fut. Ezt a címet akkor használhatod, ha az ügyfelet és a kiszolgálót egyetlen, önálló gépen teszteled.

2 Létrehozol egy InputStreamReader-t a csatoló alacsonyszintű (kapcsolati) bemeneti adatfolyamához láncolva

```
InputStreamReader stream = new InputStreamReader(chatSocket.getInputStream());
```

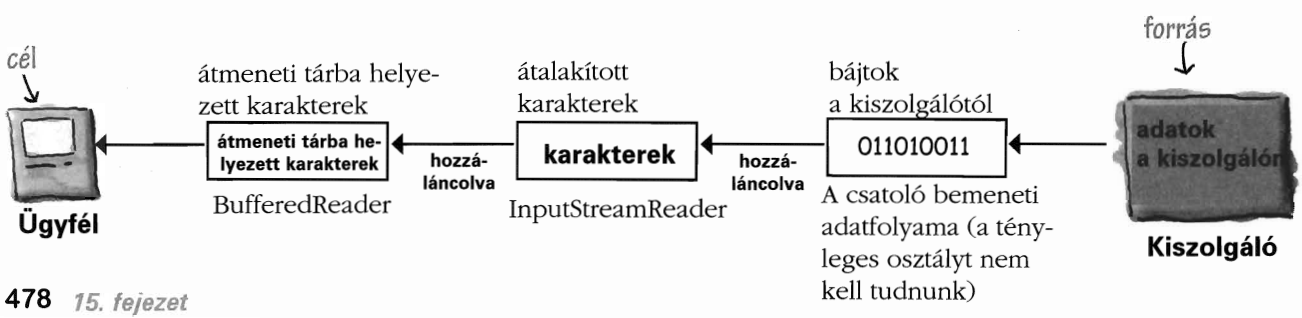
Az InputStreamReader egy „híd" egy alacsonyszintű (például a csatolóról érkező) bájtfolyam és egy magasszintű karakterfolyam (például a BufferedReader, amelyet a láncolt adatfolyamunk tetejére szeretnénk tenni) között.

Csak annyit kell tennünk, hogy KÉRÜNK a csatolótól egy bemeneti adatfolyamot. Ez egy alacsonyszintű kapcsolati adatfolyam, csak éppen hozzáláncoljuk valami máshoz, ami barátságosabb a szövegekkel.

A BufferedReader-t az InputStreamReader-hez láncoljuk (ez utóbbit pedig a csatolótól kapott alacsonyszintű kapcsolati adatfolyamhoz láncoltuk).

3 Létrehozol egy BufferedReader-t, és olvasol

```
BufferedReader reader = new BufferedReader(stream);
String message = reader.readLine();
```



Adatokat egy PrintWriter segítségével írhatsz egy csatolóra

A PrintWriter-t nem használtuk az előző fejezetben, csak a BufferedWriter-t. Itt választhatunk, de amikor egyszerre egy karakterláncot írsz ki, a PrintWriter a szokásos választás. A PrintWriter-ben ezenkívül felismerheted a két legfontosabb tagfüggvényt: a print() és a println() függvényeket. Pont mint a régi jó System.out-ban.

ez a rész ugyanaz, mint az előző oldalon – ahhoz is kapcsolódnod kell, hogy írj a kiszolgálóra

1 Létrehozol egy csatolókapcsolatot (Socket) a kiszolgálóval

```
Socket chatSocket = new Socket("127.0.0.1", 5000);
```

2 Létrehozol egy PrintWriter-t a csatoló alacsonyszintű (kapcsolati) kimeneti adatfolyamához láncolva

```
PrintWriter writer = new PrintWriter(chatSocket.getOutputStream());
```

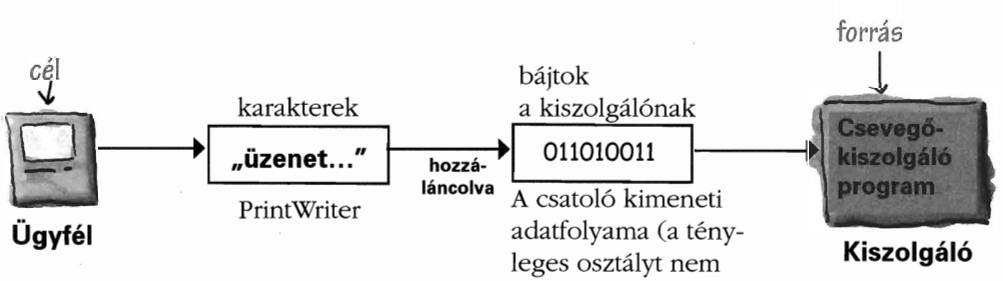
A PrintWriter a saját hídjaként működik a karakteradatok és a csatolóról érkező alacsonyszintű kimeneti adatfolyam között. A PrintWriter-t a csatoló kimeneti adatfolyamához láncolva karakterláncokat írhatunk a csatolókapcsolatra.

A csatoló ad nekünk egy alacsonyszintű kapcsolati adatfolyamot, amelyet a PrintWriter konstruktorának átadva hozzáláncolunk a PrintWriter-hez.

3 Kiírsz (print) valamit

```
writer.println("elküldeni kívánt üzenet");
writer.print("másik üzenet");
```

A println() egy új sort ad az elküldött üzenet végéhez.
A print() nem kezd új sort.



DailyAdviceClient

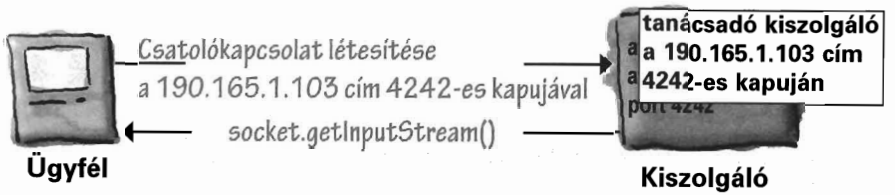
Mielőtt nekiállnánk felépíteni a csevegőprogramot, kezdjük valami kisebbel.
Az Advice Guy (Tanácsadó) egy kiszolgálói program, amely ösztönző gyakorlati tanácsokat ad, hogy átsegítsen a kódolással töltött hosszú napokon.
Az Advice Guy ügyfélprogramját fogod tehát először elkészíteni, amely minden kapcsolódáskor lehív egy üzenetet a kiszolgálóról.
Mire vársz? Ki *tudja*, milyen lehetőségeket szalasztasz el, ha nincs a kezedben egy ilyen alkalmazás!



A Tanácsadó

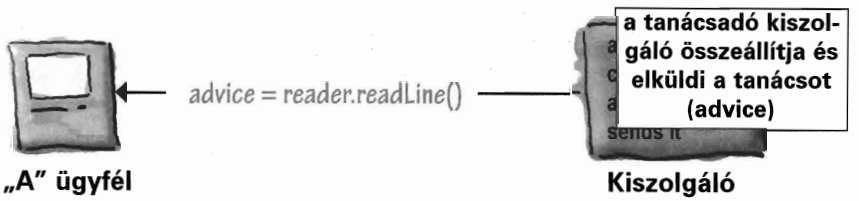
1 Kapcsolódás

Az ügyfél kapcsolódik a kiszolgálóhoz, és kap tőle egy bemeneti adatfolyamot



2 Olvasás

Az ügyfél beolvas egy üzenetet a kiszolgálóról



A DailyAdviceClient kódja

Ez a program létrehoz egy csatolót, illetve egy BufferedReader-t (más adatfolyamok segítségével), és beolvas egyetlen sort a kiszolgálói alkalmazástól (ami a 4242-es kapun fut).

```
import java.io.*;
import java.net.*;

public class DailyAdviceClient {

    public void go() {
        try {
            Socket s = new Socket("127.0.0.1", 4242);

            InputStreamReader streamReader = new InputStreamReader(s.getInputStream());
            BufferedReader reader = new BufferedReader(streamReader);

            String advice = reader.readLine();
            System.out.println("Today you should: " + advice);

            reader.close();
        } catch(IOException ex) {
            ex.printStackTrace();
        }
    }

    public static void main(String[] args) {
        DailyAdviceClient client = new DailyAdviceClient();
        client.go();
    }
}
```

A Socket osztály a java.net csomagban található.

Létrehozunk egy csatolókapcsolatot a 4242-es kapun futó dologgal (bármi is legyen az), ugyanazon a gépen, ahol ez a kód is fut (a „helyi állomáson" – localhost).

Itt sok minden félrecsúszhat.

Hozzáláncolunk egy BufferedReader-t egy InputStreamReader-hez, amely beolvassa a bementi adatfolyamot a csatolóról.

Ez a readLine() utasítás PONTOSAN ugyanolyan, mintha egy FÁJLHOZ láncolt BufferedReader-t használnánk. Más szavakkal, mire meghívod a BufferedWriter valamelyik tagfüggvényét, az író nem fogja tudni, hogy honnan származnak a karakterek, és nem is érdekli.

Ez az ÖSSZES adatfolyamot bezárja.

Hegyezd ki a ceruzád!

Teszteld a memóriádat a csatolóról olvasó és oda író adatfolyamokkal, illetve osztályokkal kapcsolatban – és ne lapozz vissza az előző oldalra!

Szöveg ***olvasása*** egy csatolóról:



Ügyfél

írás az adatfolyamok láncába, amelyet az ügyfél arra használ, hogy olvasson a kiszolgálóról

Forrás



Kiszolgáló

Szöveg ***küldése*** egy csatolóra:



Ügyfél

írás az adatfolyamok láncába, amelyet az ügyfél arra használ, hogy küldjön valamit a kiszolgálónak

Cél



Kiszolgáló

Hegyezd ki a ceruzád!

Írd be a válaszokat:

Milyen két információra van szüksége az ügyfélnek ahhoz, hogy csatolókapcsolatot hozhasson létre egy kiszolgálóval?

_____ _____

Milyen TCP-kapuszámok vannak fenntartva az olyan „jól ismert" szolgáltatások számára, mint a HTTP és az FTP?

_____

IGAZ vagy HAMIS? „Az érvényes TCP-kapuszámok tartománya egy short primitívvel ábrázolható."

_____

Egyszerű kiszolgáló készítése

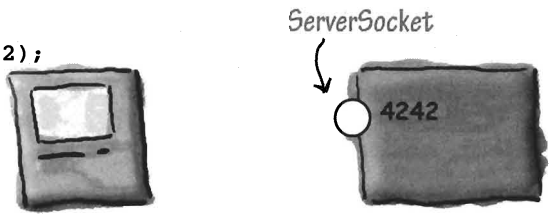
Mi kell ahhoz, hogy kiszolgálóprogramot készíthess? Mindössze pár csatoló. Igen, a pár itt *kettőt* jelent: egy ServerSocket (kiszolgálócsatoló) csatolót, amely az ügyfelek kérelmeire várakozik (amikor egy ügyfél új csatolót hoz létre), valamint egy egyszerű Socket-et, amelyet az ügyféllel való kommunikációra használunk.

Hogyan működik?

1. A kiszolgálóprogram létrehoz egy ServerSocket-et egy bizonyos kapun:

```
ServerSocket serverSock = new ServerSocket(4242);
```

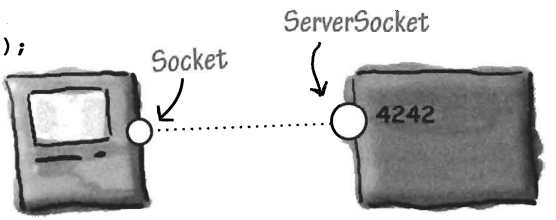
Ez elindítja a kiszolgálóprogramot, amely a 4242-es kapura érkező ügyfélkérelmeket figyeli.



2. Az ügyfél létrehoz egy csatolókapcsolatot a kiszolgálóprogramhoz:

```
Socket sock = new Socket("190.165.1.103", 4242);
```

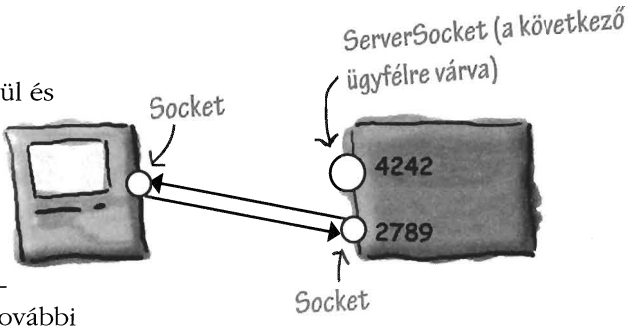
Az ügyfél ismeri az IP-címet és a kapuszámot (nyilvános forrásból, vagy mert az, aki a kiszolgálóprogramot az adott kapura állította, megadta neki).



3. A kiszolgáló létrehoz egy új Socket-et, hogy kommunikálni tudjon az adott ügyféllel:

```
Socket sock = serverSock.accept();
```

Az accept() – „elfogad" – tagfüggvény leblokkol (csak ül és vár), amíg létre nem jön egy csatolókapcsolat egy ügyféllel. Amikor végül egy ügyfél megpróbál kapcsolódni, a tagfüggvény visszaad egy egyszerű Socket-et (egy *másik* kapun), amely tudja, hogyan kommunikálhat az ügyféllel (vagyis ismeri az *ügyfél* IP-címét és kapuszámát). A Socket azért kerül a ServerSocket-tő eltérő kapura, hogy a ServerSocket közben további ügyfelekre várakozhasson.



A DailyAdviceServer kódja

Ez a kiszolgálóprogram létrehoz egy ServerSocket-et, és várja az ügyfelek kérelmeit. Amikor kérelem érkezik egy ügyféltől (vagyis az ügyfél new Socket() utasítást adott erre az alkalmazásra), a kiszolgáló létrehoz egy új csatolókapcsolatot az adott ügyféllel, majd létrehoz egy PrintWriter-t (a Socket kimeneti adatfolyamát felhasználva), és üzenetet küld az ügyfélnek.

(Ne feledd, ezeket a karakterláncokat a kódszerkesztő tördelte így. Egy karakterlánc közepén soha ne nyomj Entert!)

Ne feledkezz meg a betöltésekről.

A napi tanács ebből a tömbből származik.

```
import java.io.*;
import java.net.*;

public class DailyAdviceServer {

  String[] adviceList = {"Kisebbeket harapj", "Válaszd a szűk farmert. Nem, NEM kövérít", "Egy szó: helytelen", "Egyszer légy őszinte, és mondd meg a főnöködnek, mit gondolsz *valójában*", "Ezt a frizurát át kellene gondolnod"};

    public void go() {

        try {
            ServerSocket serverSock = new ServerSocket(4242);

            while(true) {

                Socket sock = serverSock.accept();

                PrintWriter writer = new PrintWriter(sock.getOutputStream());
                String advice = getAdvice();
                writer.println(advice);
                writer.close();
                System.out.println(advice);
            }

        } catch(IOException ex) {
            ex.printStackTrace();
        }
    } // a go lezárása

    private String getAdvice() {
        int random = (int) (Math.random() * adviceList.length);
        return adviceList[random];
    }

    public static void main(String[] args) {
        DailyAdviceServer server = new DailyAdviceServer();
        server.go();
    }
}
```

A ServerSocket arra utasítja ezt a kiszolgálóprogramot, hogy a 4242-es kapura érkező ügyfélkérelmeket figyelje azon a gépen, ahol ez a kód fut.

A kiszolgáló folyamatosan ismétli ezt a kódot az ügyfélkérelmekre várva (és kiszolgálva azokat).

Az accept() tagfüggvény blokkol (csak ül és vár), amíg kérelem nem érkezik, majd visszaad egy Socket-et (valamelyik szabad kapun) az ügyféllel való kommunikációhoz.

Az ügyféllel létesített csatolókapcsolat felhasználásával létrehozunk egy PrintWriter-t, és átadunk neki (println()) egy String formátumú tanácsot. Ezt követően bezárjuk a csatolót, mert az adott ügyféllel ezzel végeztünk.



Agygyúró

Honnan tudja a kiszolgáló, hogy hogyan kommunikálhat az ügyféllel?

Az ügyfél ismeri a kiszolgáló IP-címét és kapuszámát, de hogyan képes a kiszolgáló csatolókapcsolatot létesíteni az ügyféllel (és bemeneti-kimeneti adatfolyamokat létrehozni)?
Gondold végig, hol, hogyan és mikor szerez tudomást a kiszolgáló az ügyfélről.

Nincsenek

hülye kérdések

K: A tanácskiszolgáló kódjának az előző oldalon van egy NAGYON komoly korlátja: úgy tűnik, egyszerre csak egy ügyfelet tud kiszolgálni!

V: Igen, ez így van. Addig nem képes kérelmet fogadni egy ügyféltől, amíg az előzővel nem végzett, és el nem indította a végtelen ciklus következő ismétlését (amelyben addig várakozik az accept() hívásnál, amíg újabb kérelem nem érkezik, amikor is létrehoz egy csatolókapcsolatot az új ügyféllel, és újrakezdi az eljárást).

K: Akkor átfogalmaznám a problémát: hogyan készíthetek olyan kiszolgálót, amely párhuzamosan több ügyfelet is képes kezelni??? Ez a kód *soha* nem lenne alkalmas például csevegőkiszolgálónak.

V: A megoldás nagyon egyszerű. Külön szálakat kell használnod, és minden új ügyfélcsatolót egy új szálhoz kell rendelned – ezt fogod hamarosan megtanulni.



Újratöltve

- Az ügyfél- és kiszolgálóalkalmazások csatolókapcsolatokon keresztül kommunikálnak egymással.
- A csatoló (Socket) egy kapcsolatot jelképez két alkalmazás között, amelyek akár két különböző fizikai számítógépen is futhatnak (de ez nem feltétel).
- Az ügyfélnek ismernie kell a kiszolgálóprogram IP-címét (vagy tartománynevét) és TCP-kapuszámát.
- A TCP-kapu egy adott kiszolgálóprogramhoz rendelt, 16 bites, előjel nélküli szám. A TCP-kapuszámok lehetővé teszik, hogy ugyanahhoz a géphez különböző ügyfelek kapcsolódjanak, de az ott futó programok közül más-más alkalmazásokkal kommunikáljanak.
- A 0 és 1023 közötti kapuszámok az olyan „jól ismert szolgáltatásoknak" vannak fenntartva, mint a HTTP, az FTP vagy az SMTP.
- Egy ügyfél úgy kapcsolódhat egy kiszolgálóhoz, hogy létrehoz egy csatolókapcsolatot:

```
Socket s = new Socket("127.0.0.1", 4200);
```

- Miután kapcsolódott, az ügyfél bemeneti és kimeneti adatfolyamokat szerezhet a csatolótól, amelyek alacsonyszintű „kapcsolati" adatfolyamok:

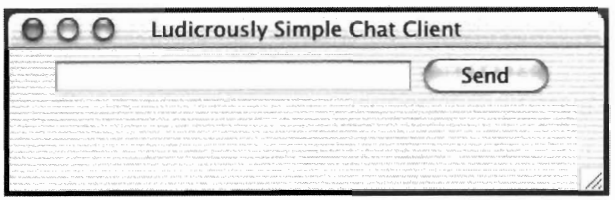
```
sock.getInputStream();
sock.getOutputStream();
```

- Ha szöveges adatokat szeretnél olvasni egy kiszolgálóról, létre kell hoznod egy BufferedReader-t, és hozzá kell láncolnod azt egy InputStreamReader-hez, amely maga a csatolóról származó bemeneti adatfolyamhoz láncolódik.
- Az InputStreamReader egy „hídként" működő adatfolyam, amely bájtokat alakít át szöveges (karakteres) adatokká. Elsősorban az a feladata, hogy középső láncszemként működjön a magasszintű BufferedReader és az alacsonyszintű bemeneti csatoló-adatfolyam között.
- Ha szöveges adatokat szeretnél írni egy kiszolgálóra, egy közvetlenül a csatoló kimeneti adatfolyamához láncolt PrintWriter-t kell létrehoznod. A kiszolgálónak a print() vagy a println() tagfüggvényt meghívva küldhetsz karakterláncokat.
- A kiszolgálók kiszolgálócsatolókat (ServerSocket) használnak, amelyek egy adott számú kapunk várakoznak az ügyfelek kérelmeire.
- Amikor egy ServerSocket-hez kérelem érkezik, az úgy „fogadja el" a kérelmet, hogy létrehoz egy csatolókapcsolatot az ügyféllel.

Csevegőügyfél készítése

A csevegőügyfelünket – Ludicrously Simple Chat Client (Nevetségesen Egyszerű Csevegőügyfél) – két lépésben fogjuk elkészíteni. Először egy csak küldésre alkalmas változatot építünk, amely üzeneteket küld a kiszolgálónak, de nem olvassa a többi résztvevő üzeneteit (ami izgalmas és rejtélyes csavar a csevegőszoba-alkalmazások elvét tekintve).
Ha ezzel megvagyunk, elkészítjük a teljes alkalmazást, amely egyaránt képes lesz üzeneteket küldeni *és* fogadni.

Első változat: csak küldés



Beírsz egy üzenetet, majd megnyomod a „Send" (Küldés) gombot, hogy elküldd a szöveget a kiszolgálónak. Ebben a változatban nem kapunk semmilyen üzenetet a kiszolgálótól, ezért nincs gördíthető szövegmező.

Kódvázlat

```
public class SimpleChatClientA {

   JTextField outgoing;
   PrintWriter writer;
   Socket sock;

   public void go() {
// felépítjük a grafikus felületet, és bejegyeztetünk egy figyelőt a send gombhoz
// meghívjuk a setUpNetworking() tagfüggvényt
   }

   private void setUpNetworking() {
// létrehozunk egy csatolót, majd egy PrintWriter-t
// a PrintWriter-t a writer példányváltozóhoz rendeljük
   }

   public class SendButtonListener implements ActionListener {
      public void actionPerformed(ActionEvent ev) {
// kiolvassuk a szöveget a szövegmezőből, és
// elküldjük a kiszolgálónak az író (egy PrintWriter) segítségével
      }
   } // lezárjuk a SendButtonListener belső osztályt

} // lezárjuk a külső osztályt
```

```
import java.io.*;
import java.net.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SimpleChatClientA {

    JTextField outgoing;
    PrintWriter writer;
    Socket sock;

    public void go() {
        JFrame frame = new JFrame("Ludicrously Simple Chat Client");
        JPanel mainPanel = new JPanel();
        outgoing = new JTextField(20);
        JButton sendButton = new JButton("Send");
        sendButton.addActionListener(new SendButtonListener());
        mainPanel.add(outgoing);
        mainPanel.add(sendButton);
        frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
        setUpNetworking();
        frame.setSize(400,500);
        frame.setVisible(true);
    } // a go lezárása

    private void setUpNetworking() {
        try {
            sock = new Socket("127.0.0.1", 5000);
            writer = new PrintWriter(sock.getOutputStream());
            System.out.println("networking established");
        } catch(IOException ex) {
            ex.printStackTrace();
        }
    } // a setUpNetworking lezárása

  public class SendButtonListener implements ActionListener {
     public void actionPerformed(ActionEvent ev) {
         try {
            writer.println(outgoing.getText());
            writer.flush();

         } catch(Exception ex) {
            ex.printStackTrace();
         }
         outgoing.setText("");
         outgoing.requestFocus();
     }
    } // a SendButtonListener belső osztály lezárása

    public static void main(String[] args) {
       new SimpleChatClientA().go();
    }
} // a külső osztály lezárása
```

Az adatfolyamok (java.io), csatolók (java.net) és GUI-cuccok betöltése

A grafikus felület felépítése – semmi újdonság nincs benne, és semmi olyan, ami a hálózatkezeléshez vagy a bemenet-kimenethez kapcsolódna.

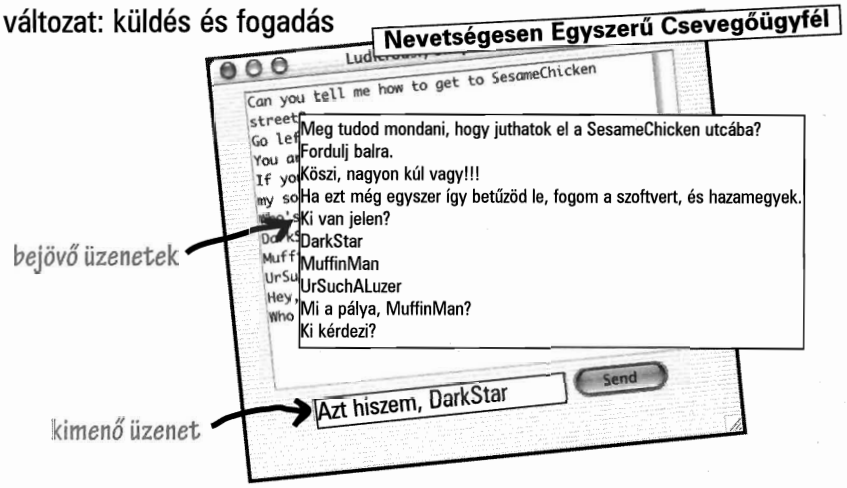
Azért helyi állomást használunk, hogy az ügyfelet és a kiszolgálót egyetlen gépen próbálhassuk ki.

Itt hozzuk létre a csatolót és a PrintWriter-t (a go() tagfüggvényből meghívva, közvetlenül a program grafikus felületének megjelenítése előtt).

Itt végezzük a tényleges írást. Ne feledd, hogy az író a csatolóról származó kimeneti adatfolyamhoz van láncolva, így minden println() utasítás áthalad a hálózaton a kiszolgáló felé.

Ha ezt most ki szeretnéd próbálni, gépeld be a csevegőkiszolgáló Készre Sütött Kódját a fejezet végéről, majd először indítsd el a kiszolgálót egy terminálban, aztán az ügyfelet egy másikban.

Második változat: küldés és fogadás



A Nagy Kérdés: HOGYAN *jutnak el hozzád az üzenetek a kiszolgálóról?*

Egyszerűen: amikor beállítod a hálózati működést, egy bemeneti adatfolyamot (valószínűleg egy BufferedReader-t) is létrehozol. Ezt követően az üzeneteket a readLine() segítségével olvashatod be.

A Még Nagyobb Kérdés: MIKOR *kapod meg az üzeneteket a kiszolgálótól?*

Gondold végig, milyen lehetőségek jöhetnek szóba:

1 *Első lehetőség: a kiszolgáló lekérdezése 20 másodpercenként*

Előnyök: Nos, kivitelezhető.

Hátrányok: Honnan tudja a kiszolgáló, hogy mi az, amit már láttál, és mi az, amit még nem? A kiszolgálónak tárolnia kell az üzeneteket, ahelyett, hogy kézbesítés után eldobhatná őket. És miért 20 másodpercenként? Egy ilyen késleltetés csökkenti a használhatóságot, ha viszont csökkented az időközt, sokszor feleslegesen fogod zaklatni a kiszolgálót, ami nem túl hatékony megoldás.

2 *Második lehetőség: akkor olvasol a kiszolgálóról, amikor a felhasználó üzenetet küld*

Előnyök: Kivitelezhető és nagyon egyszerű.

Hátrányok: Hülyeség. Miért ellenőriznéd ilyen véletlenszerűen az üzeneteket? Mi van, ha a felhasználó csak nézelődik, és soha nem küld üzenetet?

3 *Harmadik lehetőség: minden üzenetet akkor olvasol be, amikor a kiszolgáló elküldi*

Előnyök: A leghatékonyabb és leghasználhatóbb megoldás.

Hátrányok: Hogyan csinálhatsz két dolgot egyszerre? Hová teszed a kódot? Egy ciklusra lesz szükséged, ami folyamatosan vár, hogy olvashasson a kiszolgálóról. De hová kerüljön ez a ciklus? Miután elindítottad a grafikus felületet, semmi sem történik, amíg egy felületi összetevő el nem indít egy eseményt.



A Javában MEGTEHETED, hogy séta közben rágózol.

Mostanra már tudod, hogy a harmadik megoldást fogjuk választani.

Valami olyasmit szeretnénk, ami folyamatosan fut, a kiszolgálótól érkező üzeneteket figyelve, *de anélkül, hogy zavarná a felhasználót a grafikus felület használatában.* Tehát amíg a felhasználó gondtalanul gépeli az új üzeneteket, vagy görgeti a beérkezetteket, a *színfalak mögött* szeretnénk valamivel folyamatosan beolvastatni azt, ami a kiszolgálóról érkezik. Ez azt jelenti, hogy egy új szálra lesz szükségünk – egy új, különálló veremre.

Azt szeretnénk, hogy minden ugyanúgy működjön, mint a csak küldésre alkalmas (az első) változatban, de ezzel párhuzamosan egy új *folyamat* (process) is fusson, amely információkat olvas a kiszolgálóról, és megjeleníti azokat a bejövő szövegek számára fenntartott területen.

Illetve nem egészen. Hacsak nincs több processzor a számítógépedben, az új Java-szálak valójában nem önálló folyamatok az operációs rendszeren. Csak úgy *tűnik*, mintha azok lennének.

Többszálúság a Javában

A többszálúságot egyenesen beleszőtték a Java nyelv szövetébe. Új végrehajtási szálat pofonegyszerű létrehozni:

```
Thread t = new Thread();
t.start();
```

Ennyi az egész. Az új Thread (Szál) *objektum* létrehozásával önálló *végrehajtási szálat* indítasz, amely saját hívási veremmel rendelkezik.

Egy problémától eltekintve.

Az új szál nem csinál semmit, ezért lényegében abban a pillanatban „meghal", amint megszületett. Amikor pedig egy szál elpusztul, az új verem is semmivé lesz. Itt a vége, fuss el véle.

Hiányzik tehát egy kulcsfontosságú összetevő: a szál *feladata* (job). Más szavakkal, az a kód, amit önálló szálban szeretnél futtatni.

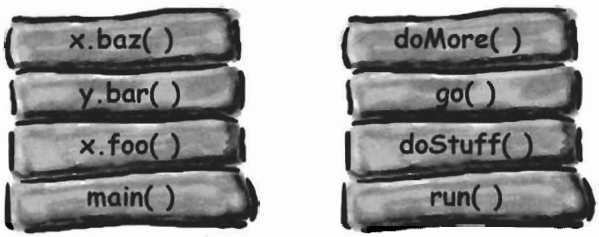
Ahhoz, hogy több szálat működtethess a Javában, a *szálak* mellett az általuk *futtatott feladatokról* is gondoskodnod kell. Ezenkívül szükséged lesz a java.lang csomagban található Thread osztályra. (Emlékezz vissza, hogy a java.lang csomag betöltése automatikusan történik, mert ebben találhatók a nyelv legfontosabb osztályai, például a String és a System.)

A Javában több szál is lehet, de csak egy Thread osztály

Amikor egy *szálról* beszélünk, egy önálló végrehajtási szálat értünk alatta, vagyis egy önálló hívási vermet. Ha viszont azt látod, hogy **Thread**, gondolj a Java elnevezési szabályaira. Mi kezdődik a Javában nagybetűvel? Az osztályok és a felületek neve. A **Thread** egy osztály a java.lang csomagban. A **Thread** objektumok egy-egy *végrehajtási szálat* jelképeznek: minden alkalommal, amikor új végrehajtási *szálat* akarsz indítani, a **Thread** osztály egy új példányát kell létrehoznod.

A szálak önálló „végrehajtási szálak", vagyis önálló hívási vermek.
A Thread a Java egyik osztálya, amely egy szálat jelképez.
Ha új szálat szeretnél indítani, egy új Thread objektumot kell létrehoznod.

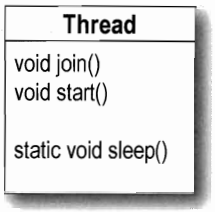
szál



fő szál

a kód által indított másik szál

Thread



a java.lang.Thread osztály

A szálak tehát önálló végrehajtási szálak, önálló hívási veremmel. Minden Java-alkalmazás elindít egy fő szálat – azt a szálat, amelyik a verem aljára helyezi a main() tagfüggvényt. A JVM felel a fő szál (illetve szükség szerint más szálak, például a szemétgyűjtő szál) elindításáért, programozóként azonban írhatsz olyan kódot, ami saját szálakat indít.

A Thread egy osztály, amely egy végrehajtási szálat jelképez. Ebben az osztályban szálak indítására, szálak összekapcsolására, illetve szálak elaltatására szolgáló tagfüggvények találhatók. (Persze vannak további tagfüggvényei is, de ezek a legfontosabbak, amelyeknek a használatára most szükséged lesz.)

Mit jelent az, ha egynél több hívási verem van?

Az egynél több hívási verem annak a *látszatát* kelti, hogy egyszerre több dolog történik. A valóságban persze csak egy valódi többprocesszoros rendszer tud párhuzamosan több dolgot csinálni, a Java-szálak használatakor azonban úgy *tűnhet*, mintha egyidejűleg különböző műveleteket végeznénk. Más szavakkal, a végrehajtás olyan gyorsan vált a vermek között, hogy úgy érezzük, mintha mindegyik verem utasításai egyszerre hajtódnának végre. Mindazonáltal ne feledd, hogy a Java csak egy folyamat, amely a háttérben működő operációs rendszeren fut, tehát először is *magának* a Javának kell „az éppen végrehajtás alatt álló folyamatnak" lennie az operációs rendszeren. Ahogy viszont a Javára kerül a sor, hogy végrehajtson valamit, *pontosan* mit is *futtat* a Java virtuális gép? Melyik bájtkódot hajtja végre? Azt, amelyik az éppen futó verem tetején található! 100 ezredmásodpercen belül pedig az éppen végrehajtás alatt álló kód egy *másik* verem *másik* tagfüggvényére válthat.
A szálak egyik feladata az, hogy nyomon kövessék, hogy melyik tagfüggvény melyik utasítása áll éppen végrehajtás alatt a szál vermében.
A dolog valahogy így fest:

1 A JVM meghívja a main() tagfüggvényt.

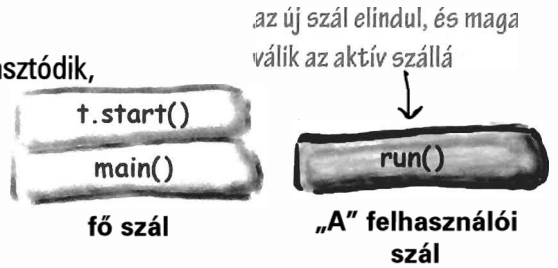
```
public static void main(String[] args) {
...
}
```



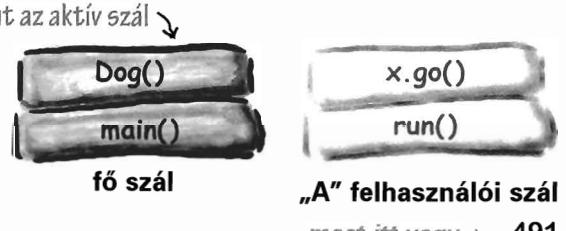
2 A main() új szálat indít. A fő szál átmenetileg lefagyasztódik, amíg az új szál fut.

```
Runnable r = new MyThreadJob();
Thread t = new Thread(r);
t.start();
Dog d = new Dog();
```

mindjárt megtudod, hogy ez mit jelent...



3 A JVM váltogatja az új szálat („A" felhasználói szál) és az eredeti fő szálat, amíg mindkettő be nem fejeződik.



Így indíthatsz új szálat:

1 **Létrehozol egy Runnable objektumot (a szál feladatát):**

```
Runnable threadJob = new myRunnable();
```

A Runnable („futtatható") egy felület, amelyről a következő oldalon lesz szó. Írsz egy osztályt, amely megvalósítja a Runnable felületet, és ez az osztály az a hely, ahol meghatározod a feladatot, amit a szálnak végre kell hajtania, vagyis azt a tagfüggvényt, amelyet a szál új hívási verméből futtatni kell.



Runnable objektum

2 **Létrehozol egy Thread objektumot (a munkát végző szálat), és átadod neki a Runnable objektumot (a feladatot):**

```
Thread myThread = new Thread(threadJob);
```

Átadod az új Runnable objektumot a Thread konstruktorának; ezzel közlöd az új Thread objektummal, hogy melyik tagfüggvényt tegye az új verem aljára – a Runnable run() tagfüggvényét.

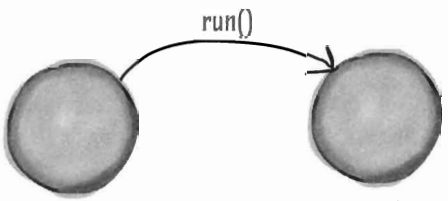


Thread objektum

3 **Elindítod a szálat:**

```
myThread.start();
```

Amíg meg nem hívod a Thread objektum start() tagfüggvényét, semmi sem történik. Ezzel a hívással jutsz el onnan, hogy van egy Thread-példányod, oda, hogy új végrehajtási szálat indítottál. Amikor az új szál elindul, fogja a Runnable objektum run() tagfüggvényét, és az új szál vermének aljára helyezi.



Thread objektum **Runnable objektum**

Minden szálnak van egy elvégzendő feladata – egy tagfüggvény, amelyet az új szál vermébe kell tenni.



A Runnable olyan egy Thread számára, mint egy elvégzendő munka egy alkalmazottnak. A Runnable objektum az a feladat, amit a szálnak el kell végeznie.
A Runnable objektum tárolja azt a tagfüggvényt, amely az új szál vermének aljára kerül: a run()-t.

A Thread objektumoknak egy feladatra van szükségük. Ezt a feladatot fogja futtatni a szál, amikor elindítod. A feladat az első tagfüggvény, amely az új szál vermébe kerül, és mindig egy ilyen formájú tagfüggvénynek kell lennie:

```
public void run() {
// az új szál által futtatandó kód
}
```

A Runnable felület csak egy tagfüggvényt – public void run() – határoz meg. (Ne feledd, hogy felületről van szó, ezért a tagfüggvény attól függetlenül nyilvános lesz, hogy így jelölöd-e meg.)

Honnan tudja a szál, hogy melyik tagfüggvényt kell a verem aljára tennie? Onnan, hogy a Runnable egy szerződést határoz meg. Onna, hogy a Runnable egy felület. Egy szál feladata bármely olyan osztályban meghatározható, amely megvalósítja a Runnable felületet. A szálat csak az érdekli, hogy a Thread konstruktorának egy olyan osztályból származó objektumot adj át, amelyik megvalósítja a Runnable felületet.
Amikor egy Thread konstruktornak átadsz egy Runnable objektumot, valójában csak módot adsz a szálnak arra, hogy hozzáférjen egy run() tagfüggvényhez – vagyis megadod azt a feladatot, amelyet a szálnak el kell végeznie.

Ahhoz, hogy feladatot adj egy szálnak, meg kell valósítanod a Runnable felületet

A Runnable a java.lang csomagban található, ezért nem kell külön betöltened.

```
public class MyRunnable implements Runnable {

    public void run() {
        go();
    }

    public void go() {
        doMore();
    }

    public void doMore() {
        System.out.println("top o' the stack");
    }
}
```

A Runnable felületnek csak egy megvalósítandó tagfüggvénye van: a public void run() (argumentumok nélkül). Itt kell elhelyezned a FELADATOT, amelyet a szálnak futtatnia kell. Ez a tagfüggvény kerül az új verem aljára.

"a verem teteje"

2

```
class ThreadTester {

    public static void main (String[] args) {

        Runnable threadJob = new MyRunnable();
        Thread myThread = new Thread(threadJob);

        myThread .start();

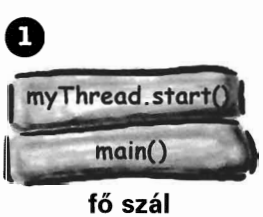
        System.out.println("back in main");
    }
}
```

Átadod az új Runnable-példányt az új Thread konstruktornak. Ez közli a szállal, hogy melyik tagfüggvényt kell az új verem aljára tennie, vagyis hogy melyik lesz az első tagfüggvény, amelyet az új szál futtat.

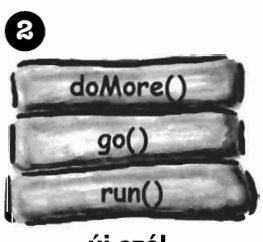
1

Addig nem kapsz új végrehajtási szálat, amíg meg nem hívod a start()-ot a Thread-példányra: egy szál addig nem is igazán szál, amíg el nem indítod, csupán egy Thread-példány – olyan, mint bármely más objektum, igazi „szálság" nélkül.

"ismét a main-ben"



fő szál



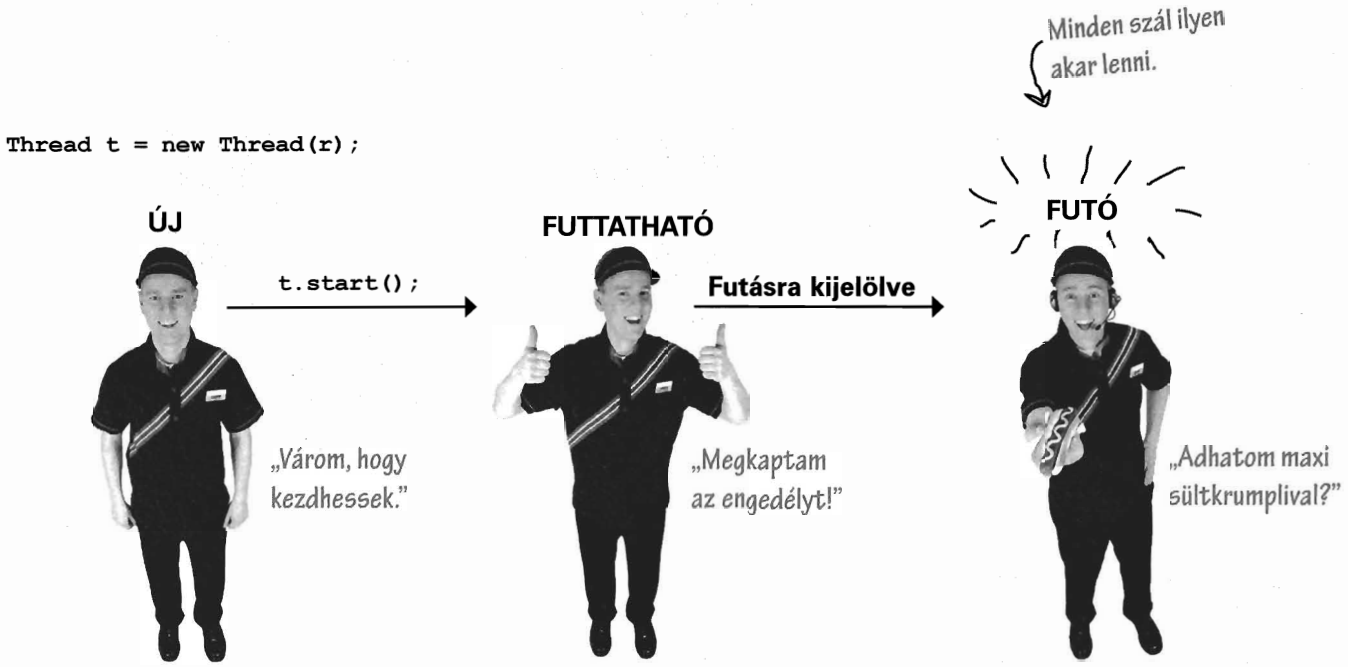
új szál



Agygyúró

Mit gondolsz, milyen kimenetet kapsz, ha futtatod a ThreadTester osztályt? (Néhány oldalon belül megtudod.)

Egy új szál három állapota



`Thread t = new Thread(r);`

A Thread-példány létrejött, de még nem lett elindítva. Más szavakkal, van egy Thread *objektumod*, de nincs *végrehajtási szálad*.

`t.start();`

Amikor elindítod a szálat, az futtatható állapotba kerül. Ez azt jelenti, hogy a szál készen áll a futásra, és várja a Nagy Lehetőséget: azt, hogy kijelöljék végrehajtásra. Ezen a ponton a szálnak már van egy új hívási verme.

Ez az az állapot, ami után minden szál vágyakozik. Hogy ő lehessen a Kiválasztott, az Éppen Futó Szál. Ezt a döntést azonban csak a JVM szálütemezője hozhatja meg. A döntést néha *befolyásolhatod*, de nem kényszerítheted ki, hogy egy szál futtathatóból futó állapotba kerüljön. A futó állapotban a szál (és CSAK az adott szál) hívási verme aktív, és a verem tetején található tagfüggvény áll végrehajtás alatt.

De tovább is van: miután a szál futtatható lett, oda-vissza válthat a futtatható, a futó és még egy állapot, az *átmenetileg nem futtatható* (más néven „blokkolt") állapot között.

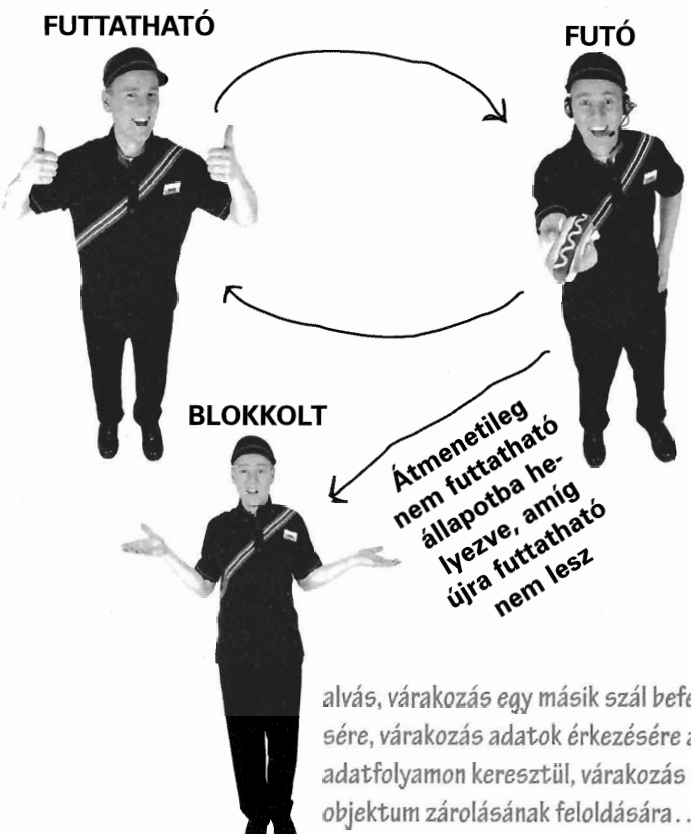
Szokványos futtatható-futó ciklus

Egy szál jellemzően a futtatható és a futó állapot között vált oda-vissza, ahogy a JVM szálütemezője kiválaszt egy szálat, majd visszalöki, hogy egy másik szálnak is esélyt adjon.



Egy szál átmenetileg nem futtathatóvá is tehető

A szálütemező egy futó szálat blokkolt állapotba is helyezhet, mégpedig különböző okokból. Előfordulhat például, hogy a szál olyan kódot hajt végre, amely egy csatoló bemeneti adatfolyamából olvas, de éppen nincsenek beolvasható adatok. Az ütemező ilyenkor kiveszi a szálat a futó állapotból, amíg adatok nem érkeznek. Az is lehet, hogy a végrehajtás alatt álló kód utasítja alvásra – sleep() – a szálat, vagy a szál azért vár, mert megpróbált meghívni egy tagfüggvényt egy objektumra, de az objektum éppen „zárolás"alatt állt. Ebben az esetben a szál addig nem folytatódhat, amíg az objektum zárolását fel nem oldja a zárat birtokló szál.
Ezek (és más körülmények is) okozhatják, hogy egy szál átmenetileg nem futtatható állapotba kerül.



A szálütemező

Mindig a szálütemező dönti el, hogy melyik futtatható szálból lesz futó szál, illetve hogy mikor (és milyen körülmények között) hagyja el egy szál a futó állapotot. Az ütemező dönti el, melyik szál futhat és mennyi ideig, és hogy a szálak hová kerülnek, amikor az ütemező úgy dönt, hogy kirúgja őket a futó állapotból. Az ütemező felett nincs hatalmad. Nincs alkalmazásprogramozási felület, amelynek a tagfüggvényeit meghívhatnád az ütemezőre, és ami a legfontosabb: az ütemezésre nézve nincsenek garanciák. (Van néhány „majdnem garancia", de azok is ködösek kissé.) A tanulság ez: ***ne alapozd a programjaid helyes működését arra, hogy az ütemező egy bizonyos módon működik!***

Az ütemezőt a különböző JVM-ek más-más módon valósítják meg, ezért még ugyanannak a programnak ugyanazon a gépen való futtatása is eltérő eredménnyel járhat. Az egyik legnagyobb hiba, amit egy kezdő Java-programozó elkövethet, hogy a többszálú programjait egyetlen számítógépen teszteli, és feltételezi, hogy a szálütemező mindig ugyanúgy fog működni, függetlenül attól, hogy hol fut a program. Mit jelent mindez az „egyszer megírod, mindenhol futtathatod" elv szempontjából? Nos, azt jelenti, hogy a többszálú Java-programjaid csak akkor lesznek valóban rendszerfüggetlenek, ha a szálütemező viselkedésétől függetlenül működnek. Azt jelenti, hogy nem támaszkodhatsz például arra, hogy az ütemező gondoskodik róla, hogy minden szál pontosan ugyanannyi lehetőséget kapjon a futásra. Bár manapság igen valószínűtlen, azért előfordulhat, hogy a programod egy olyan JVM-en fut, amelynek az ütemezője ezt mondja: „Oké, ötös szál, te jössz – és amíg rajtam múlik, addig futhatsz, amíg kész nem vagy, vagyis amíg be nem fejezted a run() tagfüggvényedet."

A titok szinte mindig az *alvásban* rejlik. Igen, az *alvásban*. Ha egy éppen futó szálat akár csak néhány ezredmásodpercre is elaltatsz, a szál arra kényszerül, hogy elhagyja a futó állapotot, és így lehetőséget ad egy másik szálnak a futásra. A szálak sleep() – alszik – tagfüggvénye *egy valamit* garantál: egy alvó szál addig *nem* válhat ismét futó szállá, amíg az alvási ideje le nem telt. Ha egy szálat például arra utasítasz, hogy két másodpercig (2000 ezredmásodpercig) aludjon, a szál nem lehet ismét futó szál, csak *az után*, hogy a két másodperc letelt.



A szálütemező dönti el, hogy ki futhat és ki nem. Általában igazságosan váltogatja a szálakat, de erre nézve nincs garancia. Előfordulhat, hogy az egyik szálat elkényezteti, míg a többieket „éhezésre" kárhoztatja.

Egy példa arra, hogy milyen kiszámíthatatlan lehet az ütemező...

Ha ezt a kódot egyetlen gépen futtatjuk...

```
public class MyRunnable implements Runnable {

   public void run() {
      go();
   }

   public void go() {
      doMore();
   }

   public void doMore() {
      System.out.println("top o' the stack");
   }
}


class ThreadTestDrive {

   public static void main (String[] args) {

      Runnable threadJob = new MyRunnable();
      Thread myThread = new Thread(threadJob);

      myThread.start();

      System.out.println("back in main");
   }
}
```

"a verem teteje"

"ismét a main-ben"

Figyeld meg, hogy a sorrend véletlenszerűen változik. Néha az új szál végez először, néha azonban a fő szál.

Ezt a kimenetet kapjuk:

```
File Edit Window Help PickMe
% java ThreadTestDrive
back in main
top o' the stack
% java ThreadTestDrive
top o' the stack
back in main
% java ThreadTestDrive
top o' the stack
back in main
% java ThreadTestDrive
top o' the stack
back in main
% java ThreadTestDrive
top o' the stack
back in main
% java ThreadTestDrive
top o' the stack
back in main
% java ThreadTestDrive
top o' the stack
back in main
% java ThreadTestDrive
back in main
top o' the stack
```

Hogyan kaphattunk különböző eredményeket?

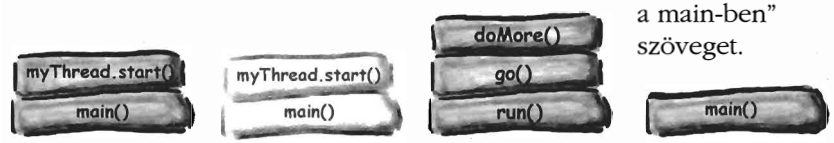
A kód néha így fut:

A main() elindítja az új szálat.

Az ütemező futóból futtatható állapotba állítja vissza a fő szálat, hogy az új szál futhasson.

Az ütemező hagyja, hogy az új szál befejezze a munkáját, és kiírja „a verem teteje" szöveget.

Az új szál eltűnik, mert a run() tagfüggvénye befejezte a feladatát. Ismét a fő szál lesz a futó szál, és kiírja az „ismét a main-ben" szöveget.



fő szál | fő szál | új szál | fő szál

idő

Néha viszont így:

A main() elindítja az új szálat.

Az ütemező futóból futtatható állapotba állítja vissza a fő szálat, hogy az új szál futhasson.

Az ütemező hagyja, hogy az új szál fusson egy ideig, de nem ad elég időt arra, hogy a run() tagfüggvény befejeződhessen.

Az ütemező futtatható állapotba állítja vissza az új szálat.

Az ütemező ismét a fő szálat választja ki futásra, az pedig kiírja az „ismét a main-ben" szöveget.

Az új szál visszaáll futó állapotba, és kiírja „a verem teteje" szöveget.



fő szál | fő szál | új szál | új szál | fő szál | új szál

idő

Nincsenek
hülye kérdések

K: Már láttam arra is példát, hogy valaki nem különálló Runnable-megvalósítást használt, hanem csak származtatott egy alosztályt a Thread-ből, és felülbírálta a Thread osztály run() tagfüggvényét. Így egy új szál létrehozásához meg lehet hívni a Thread argumentum nélküli konstruktorát: Thread t = new Thread(); // nincs Runnable

V: Igen, így *is* létrehozhatsz saját szálat, de gondold végig a dolgot az objektumközpontúság szempontjából. Mi a célja egy alosztály származtatásának? Emlékezz vissza, hogy itt két különböző dologról beszélünk – a *Thread* osztályról és a szál *feladatáról*. Az objektumközpontú tervezés felől megközelítve a kettő teljesen más, ezért külön-külön osztályba tartozik. A Thread osztály bővítésére (alosztály származtatására) csak akkor lehet szükség, ha a Thread-nek egy új, konkrétabb típusát szeretnéd létrehozni. Más szavakkal, ha a Thread-re úgy gondolsz, mint a munkát elvégző objektumra, akkor a Thread osztályt csak akkor szabad bővítened, ha konkrétabb *munkavégző* műveletekre van szükséged. Ha viszont csak egy új *feladatot* szeretnél elvégeztetni egy Thread (vagyis egy munkát végző) objektummal, akkor a Runnable felületet kell megvalósítanod egy önálló, a *feladathoz* (és nem a *munkát végző objektumhoz*) kötött osztályban.
Ez tervezési, és nem nyelvi vagy a teljesítményt érintő kérdés. Teljesen szabályos alosztályt származtatni a Thread-ből, és felülbírálni a run() tagfüggvényt – de ritkán jó ötlet.

K: Újra lehet hasznosítani egy Thread objektumot? Adhatok neki új feladatot, és újraindíthatom a start() újbóli meghívásával?

V: Nem. Miután egy szál run() tagfüggvénye befejeződött, a szált soha többé nem lehet újraindítani. Valójában a szál ekkor egy olyan állapotba kerül, amelyről eddig nem beszéltünk: ***halott*** lesz. A holt állapotban a szál befejezte a run() tagfüggvényét, és többé nem indítható újra. A Thread objektum továbbra is a kupacban lehet, olyan élő objektumként, amelyre (szükség esetén) meg lehet hívni más tagfüggvényeket, de végérvényesen elveszti a „szálságát". Más szavakkal, nincs többé önálló hívási verme, és a Thread objektum többé nem *szál*, csupán egy ugyanolyan objektum, mint a többi.
Léteznek ugyanakkor tervezési minták szálgyűjtők (pool) létrehozására, amelyeknek a segítségével továbbra is végrehajthatsz különböző feladatokat. Ezt azonban nem egy holt szál újraindításával oldhatod meg.

Újratöltve

- A szálak önálló végrehajtási szálak a Javában.
- A Javában minden szálnak saját hívási verme van.
- A Thread osztály a java.lang csomagban található. A Thread objektumok egy-egy végrehajtási szálat jelképeznek.
- Minden szálnak szüksége van egy elvégzendő feladatra. A szálak feladatai olyan osztályokból létrehozott példányok, amelyek megvalósítják a Runnable felületet.
- A Runnable felületnek csak egy tagfüggvénye van, a run(). Ez a tagfüggvény kerül az új hívási verem aljára, vagyis ez lesz az első tagfüggvény, amely az új szálban lefut.
- Új szál indításához egy Runnable objektumot kell átadnod a Thread konstruktorának.
- A szál ÚJ állapotban van, ha már példányosítottál egy Thread objektumot, de még nem hívtad meg a start() tagfüggvényt.
- Amikor elindítasz egy szálat (a Thread objektum start() tagfüggvényének meghívásával), egy új verem jön létre, amelynek az aljára a Runnable felület run() tagfüggvénye kerül. A szál ekkor FUTTATHATÓ állapotban van, és arra vár, hogy kijelöljék futásra.
- Egy szál akkor vált FUTÓ állapotra, amikor a JVM szálütemezője kijelöli futásra. Egy egyprocesszoros gépen egyidejűleg csak egy szál futhat.
- Egy szál FUTÓ állapotból néha BLOKKOLT (átmenetileg nem futtatható) állapotba kerülhet. Ennek oka lehet az, hogy a szál adatokra vár egy adatfolyamból, elaltatták, vagy mert egy objektum zárolásának feloldására vár.
- A szálütemező viselkedése kiszámíthatatlan, ezért nem támaszkodhatsz arra, hogy a szálak katonás rendben váltják egymást. A szálváltást ugyanakkor befolyásolhatod, ha a szálakat rendszeresen elaltatod.

Szálak elaltatása

Az egyik legjobb módszer, amivel biztosíthatod, hogy a szálak váltogassák egymást, a szálak rendszeres elaltatása. Ehhez csak annyit kell tenned, hogy meghívod a statikus sleep() tagfüggvényt, átadva neki az alvás időtartamát ezredmásodpercben. Például:



```
Thread.sleep(2000);
```

Ez a szálat kiüti a futó állapotból, és két másodpercig futtatható sem lesz, tehát *nem* válhat ismét futó szállá, amíg legalább két másodperc el nem telt. Némileg sajnálatos módon a sleep tagfüggvény egy ellenőrzött InterruptedException kivételt vált ki, ezért minden sleep hívást egy try/catch blokkba kell burkolnod (vagy be kell jelentened, hogy lebukhass előle). Egy sleep hívás tehát valójában így néz ki:

```
try {
  Thread.sleep(2000);
} catch(InterruptedException ex) {
  ex.printStackTrace();
}
```

Ha akarod, elaltathatsz egy szálat, hogy lehetőséget adj a többi szálnak is a futásra. Amikor egy szál felébred, mindig futtatható állapotba áll vissza, és megvárja, amíg a szálütemező ismét kijelöli futásra.

A szál álmát valószínűleg *soha* nem szakítják meg – a kivétel csak azért kapott helyet az API-ban, hogy egy olyan szálkommunikációs eljárást támogasson, amit a valóságban szinte senki nem használ. A „kezeld vagy bukj le" szabályt azonban ennek ellenére be kell tartanod, úgyhogy szokj hozzá, hogy a sleep() hívásokat egy try/catch blokkba kell csomagolnod.

Most már tudod, hogy a szálad nem fog felébredni a megadott idő letelte *előtt*, de előfordulhat, hogy valamikor az „időzítő" lejárta *után* felébred? Igen is, meg nem is. Valójában nem számít, mert amikor egy szál felébred, ***mindig a futtatható állapotba áll vissza***. A szál nem fog automatikusan felébredni és azonnal futni a megadott időben – amikor felébred, ismét a szálütemező kezébe kerül a sorsa. Azokban az alkalmazásokban, amelyek nem igényelnek tökéletes időzítést, és csak néhány száluk van, úgy tűnhet, mintha a szál az ébredés után rögtön folytatná a futását az időzített időpontban (mondjuk 2000 ezredmásodperc után), de erre nem teheted fel a programod sorsát.

A program kiszámíthatóbbá tétele a sleep segítségével

Emlékszel a korábbi példánkra, amely minden futtatásakor más eredményt adott? Lapozz vissza, és tanulmányozd át a kódot, illetve a mintakimenetet. A main-nek néha várnia kellett, amíg az új szál befejezte a munkáját (és kiírta „a verem teteje" szöveget), míg máskor az ütemező még a feladat befejezése előtt visszaküldte az új szálat futtatható állapotba, így a fő szál visszatérhetett, és kiírhatta az „ismét a main-ben" szöveget. Hogyan javíthatnánk ki a kódot? Állj meg egy pillanatra, és válaszold meg a következő kérdést: „Hol helyezhetünk el egy sleep() hívást, hogy biztosítsuk, hogy az 'ismét a main-ben' mindig 'a verem teteje' előtt íródjon ki?". Megvárjuk, amíg rálelsz egy működőképes megoldásra (amiből több is lehet).
Sikerült?

Ezt szeretnénk: a print utasítások egységes sorrendben hajtódnak végre.

```
File Edit Window Help SnoozeButton
% java ThreadTestDrive
back in main
top o' the stack
% java ThreadTestDrive
back in main
top o' the stack
% java ThreadTestDrive
back in main
top o' the stack
% java ThreadTestDrive
back in main
top o' the stack
% java ThreadTestDrive
back in main
top o' the stack
```

```
public class MyRunnable implements Runnable {

   public void run() {
      go();
   }

   public void go() {
      try {
        Thread.sleep(2000);
      } catch(InterruptedException ex) {
        ex.printStackTrace();
      }
      doMore();
   }

   public void doMore() {
      System.out.println("top o' the stack");
   }
}

class ThreadTestDrive {
   public static void main (String[] args) {
      Runnable theJob = new MyRunnable();
      Thread t = new Thread(theJob);
      t.start();
      System.out.println("back in main");
   }
}
```

A sleep meghívása itt arra kényszeríti az új szálat, hogy elhagyja a futó állapotot.

Ismét a fő szál válik a futó szállá, és kiírja az „ismét a main-ben" szöveget. Ezt követően van egy kis szünet (mintegy két másodpercnyi), mielőtt eljutunk ehhez a sorhoz, ami meghívja a doMore()-t, és kiírja „a verem teteje" szöveget.

"a verem teteje"

"ismét a main-ben"

Két szál létrehozása és elindítása

A szálaknak nevük van. Egy szálnak adhatsz saját magad által választott nevet, de elfogadhatod az alapértelmezett nevet is. Ami azonban igazán klassz a nevekkel kapcsolatban, az az, hogy a segítségükkel megmondhatod, hogy melyik szál fut. A következő példakód két szálat indít el. Mindkét szálnak ugyanaz a feladata: egy ciklust futtat, és minden ismétléskor kiírja az éppen futó szál nevét.

```
public class RunThreads implements Runnable {

    public static void main(String[] args) {
        RunThreads runner = new RunThreads();
        Thread alpha = new Thread(runner);
        Thread beta = new Thread(runner);
        alpha.setName("Alpha thread");
        beta.setName("Beta thread");
        alpha.start();
        beta.start();
    }

    public void run() {
        for (int i = 0; i < 25; i++) {
            String threadName = Thread.currentThread().getName();
            System.out.println(threadName + " is running");
        }
    }
}
```

Létrehozunk egy Runnable-példányt.

Létrehozunk két szálat, ugyanazzal a Runnable-példánnyal (vagyis ugyanazzal a feladattal – a „két szál, egy Runnable" megoldásról pár oldallal később részletesebben is szó lesz).

Nevet adunk a szálaknak.

Elindítjuk a szálakat.

Mindkét szál ezt a ciklust futtatja, minden ismétléskor kiírva a nevét.

Részlet a kimenetből, a ciklus 25 ismétlésével.

```
File Edit Window Help Centauri
Alpha thread is running
Alpha thread is running
Alpha thread is running
Beta thread is running
Alpha thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Alpha thread is running
```

Mi fog történni?

A szálak váltogatják majd egymást? A szálak neve felváltva szerepel majd? Milyen gyakran kerül sor váltásra? Minden ismétléskor? Öt ismétlés után?

Már tudod a választ: *nem tudjuk*! Minden az ütemezőtől függ. A te operációs rendszereden, a te Java virtuális gépeden, a te processzoroddal egészen más eredményt kaphatsz.

Ha a kódot OS X 10.2 (Jaguar) rendszeren, öt vagy kevesebb ismétléssel futtatod, előbb az alfa szál végzi el a munkáját, majd a béta szál fut addig, amíg végez a feladatával. Nagyon következetesen.

Nem teljes bizonyossággal, de nagyon következetesen.

Ha a ciklust azonban 25 vagy több ismétléssel futtatod, már nem ilyen tiszta a kép. Előfordulhat, hogy az alfa szál nem tudja befejezni mind a 25 ismétlést, mielőtt az ütemező visszaállítaná futtatható állapotba, és esélyt adna a béta szálnak.



Nos, öööö... igen. LÉTEZIK sötét oldal.

A szálak párhuzamossági „problémákhoz" vezethetnek.

A párhuzamossági problémák versenyhelyzetet idéznek elő, a versenyhelyzetek az adatok sérülésével járnak, az adatok sérülése félelmet kelt, a félelem pedig... a többit tudod.

Minden egyetlen halálos veszélyt rejtő helyzetre vezethető vissza: egy objektum *adataihoz* két vagy több szál férhet hozzá. Másként fogalmazva, két különböző verem tagfüggvényei a kupac ugyanazon objektumára hívnak meg kiolvasó vagy beállító utasításokat.

Ez éppen olyan, mint amikor „a jobb kéz nem tudja, mit csinál a bal". Két szál, mit sem törődve a világ többi részével, gondtalanul fütyörészve végrehajtja a tagfüggvényeit, és közben azt gondolja, hogy ő A Kiválasztott Szál. Az Egyetlen. Végül is, amikor egy szál nem fut, csak futtatható (vagy blokkolt) állapotban leledzik, lényegében öntudatlan. Amikor pedig ismét futó szál válik belőle, nem lesz tudatában, hogy valaha is megállították.

Bajba jutott házasság

Megmenthető ez a pár?

A Dr. Steve Show különkiadásában



[Átirat a #42-es epizódból]

Köszönti önöket a Dr. Steve Show!
Mai műsorunk fókuszában a két leggyakoribb ok áll, ami egy partnerkapcsolat felbomlásához vezethet: a pénz és az alvás. Mai bajba jutott vendégeink, Ryan és Monica, nem csak az ágyukat, hanem a bankszámlájukat is megosztják egymással – de már nem sokáig, hacsak nem találunk megoldást a problémájukra. Hogy mi ez a probléma? A klasszikus hiba: „két ember, egy bankszámla".
Monica így írja le a helyzetet:
„Ryan és én megegyeztünk, hogy egyikünk sem fog több pénzt kivenni a számlánkról, mint amennyi rajta van. Ezért aztán azt csináljuk, hogy aki pénzt szeretne kivenni, annak ellenőriznie *kell* a számlaegyenleget, *mielőtt* kivenné a pénzt. Azt hittük, így nagyon egyszerű lesz. Aztán hirtelen fedezetlen csekkekkel kellett küzdenünk, és állandóan túllépjük a hitelkeretünket!
Azt hittem, ez nem fordulhat elő. Hogy a megoldásunk biztonságos. De akkor *ez* történt:
Ryan-nek szüksége volt 50 dollárra, ezért ellenőrizte a számlaegyenleget, és látta, hogy a számlán 100 dollár van. Vagyis semmi gond, leemelheti a pénzt. **Csakhogy közben elaludt!**
Ekkor jöttem *én*. Amíg Ryan aludt, ki akartam venni 100 dollárt. Megnéztem az egyenleget, és mivel 100 dollár volt a számlán (ugyanis Ryan előbb aludt el, minthogy leemelte volna róla a pénzt), azt hittem, minden rendben. Kivettem a pénzt, és nem is volt semmi gond. De aztán Ryan felébredt, ő is befejezte a *saját* tranzakcióját, és máris túlterheltük a számlát! Azt se tudta, hogy elaludt, ezért aztán nyugodtan folytatta a műveletet, anélkül, hogy megint ellenőrizte volna az egyenleget. Segítsen, Dr. Steve!"
Van megoldás? Vagy a kapcsolatuknak vége? Nem akadályozhatjuk meg, hogy Ryan elaludjon, de biztosíthatjuk-e azt, hogy Monica ne nyúlhasson a bankszámlához, amíg Ryan fel nem ébredt?
Álljanak meg néhány percre, és gondolják végig a kérdést – a reklám után visszajövünk.



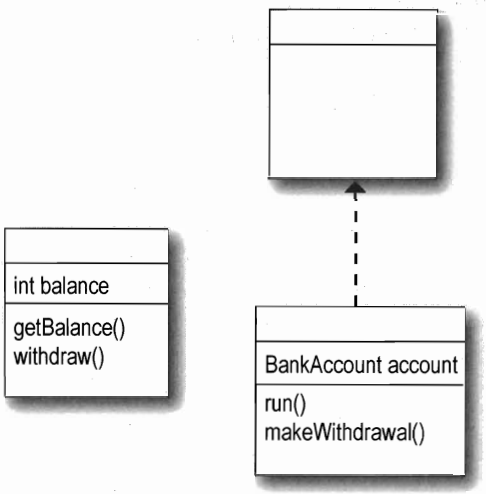
Ryan és Monica: a „két ember, egy számla" áldozatai.



Ryan elalszik, miután megnézte az egyenleget, de az előtt, hogy leemelte volna a pénzt. Amikor felébred, rögtön leemeli a pénzt, anélkül, hogy ismét ellenőrizné az egyenleget.

Ryan és Monica problémája, kódban elmesélve

A következő példa megmutatja, mi történhet, ha *két* szál (Ryan és Monica) *egyetlen* objektumon (a bankszámlán) osztozik.
A kódban két osztály található: a BankAccount (bankszámla) és a RyanAndMonicaJob (Ryan és Monica feladata).
A RyanAndMonicaJob osztály valósítja meg a Runnable felületet, és képviseli azt a viselkedést, amit Ryan és Monica is követ: az egyenleg ellenőrzését és a számla megterhelését. Természetesen azonban mindkét szál elalszik az egyenleg ellenőrzése és a pénz tényleges leemelése *között*.
A RyanAndMonicaJob osztály egy BankAccount típusú példányváltozóval rendelkezik, amely a közös számlát jelképezi.
A kód így működik:



A run() tagfüggvényben pontosan azt csináljuk, amit Ryan és Monica tenne: ellenőrizzük az egyenleget, és ha a számlán van elég pénz, megterheljük. Ez elvileg védelmet kellene, hogy nyújtson a számla túlterhelése ellen. Csakhogy... Ryan és Monica mindig elalszik, miután ellenőrizte az egyenleget, de mielőtt leemelte volna a pénzt.

1. *Létrehozzuk a RyanAndMonicaJob egy példányát.*
   A RyanAndMonicaJob osztály a Runnable objektum (az elvégzendő feladat). Mivel Monica és Ryan ugyanazt csinálja (ellenőrzi az egyenleget, és leemeli a pénzt), csak egy példány szükséges:

```
RyanAndMonicaJob  theJob = new RyanAndMonicaJob();
```

2. *Létrehozunk két szálat ugyanazzal a Runnable objektummal* ***(a RyanAndMonicaJob-példánnyal):***

```
Thread one = new Thread(theJob);
Thread two = new Thread(theJob);
```

3. *Elnevezzük és elindítjuk a szálakat.*

```
one.setName("Ryan");
two.setName("Monica");
one.start();
two.start();
```

4. *Figyeljük, ahogy mindkét szál végrehajtja a run() tagfüggvényét* ***(ellenőrzi az egyenleget, és leemeli a pénzt).***
   Az egyik szál Ryant, a másik Monicát jelképezi. Mindkét szál folyamatosan ellenőrzi az egyenleget, majd végrehajt egy pénzkivétet – de csak akkor, ha biztonságos!

```
if (account.getBalance() >= amount) {
   try {
      Thread.sleep(500);
   } catch(InterruptedException ex) {ex.printStackTrace(); }
}
```

Ryan és Monica példája

```
class BankAccount {
   private int balance = 100;
   public int getBalance() {
      return balance;
   }
   public void withdraw(int amount) {
      balance = balance - amount;
   }
}

 public class RyanAndMonicaJob implements Runnable {

   private BankAccount account = new BankAccount();

   public static void main (String [] args) {
      RyanAndMonicaJob  theJob = new RyanAndMonicaJob();
      Thread one = new Thread(theJob);
      Thread two = new Thread(theJob);
      one.setName("Ryan");
      two.setName("Monica");
      one.start();
      two.start();
    }

  public void run() {
   for (int x = 0; x < 10; x++) {
      makeWithdrawl(10);
      if (account.getBalance() < 0) {
        System.out.println("Overdrawn!");
      }
    }
 }

  private void makeWithdrawal(int amount) {
     if (account.getBalance() >= amount) {
     System.out.println(Thread.currentThread().getName() + " is about to withdraw");
      try {
        System.out.println(Thread.currentThread().getName() + " is going to sleep");
        Thread.sleep(500);
      } catch(InterruptedException ex) {ex.printStackTrace(); }
      System.out.println(Thread.currentThread().getName() + " woke up.");
      account.withdraw(amount);
      System.out.println(Thread.currentThread().getName() + " completes the withdrawl");
      }
    else {
      System.out.println("Sorry, not enough for " + Thread.currentThread().getName());
    }
  }
}
```

A számla 100 dolláros egyenlegről indul.

A RyanAndMonicaJob-nak csak EGYETLEN példánya lesz, ami EGYETLEN bankszámla-példányt jelent. Mindkét szál ezt a számlát fogja elérni.

A Runnable (a feladat) példányosítása

Létrehozunk két szálat, és átadjuk nekik ugyanazt a Runnable feladatot. Ez azt jelenti, hogy mindkét szál a Runnable osztály egyetlen számla-példányváltozójához fog hozzáférni.

A run() tagfüggvényben a szálak egy ciklust futtatnak, és minden ismétléskor megpróbálják megterhelni a számlát. Miután leemelték a pénzt, még egyszer ellenőrzik az egyenleget, hogy lássák, nincs-e túlterhelve a számla.

Ellenőrizzük a számlaegyenleget, és ha nincs elég pénz a számlán, csak egy üzenetet írunk ki. Ha VAN elég pénz, aludni térünk, majd felébredünk, és befejezzük a terhelést, ahogy Ryan is tette.

Beszúrunk egy csomó print utasítást, hogy lássuk, mi történik futás közben.



Hogy történhetett ez?

A makeWithdrawal() tagfüggvény mindig ellenőrzi az egyenleget, mielőtt pénzt emelne le (withdrawal), de így is túlterheljük a számlát. Íme egy forgatókönyv:

Ryan ellenőrzi az egyenleget, látja, hogy a számlán elég pénz van, majd álomba merül.

Közben megérkezik Monica, és ő is megnézi az egyenleget. Ő is azt látja, hogy van elég pénz, és fogalma sincs róla, hogy amikor majd Ryan felébred, meg fogja terhelni a számlát.

Monica elalszik, Ryan viszont felébred, és befejezi az általa megkezdett tranzakciót. Ezt követően Monica is magához tér, és ő is megterheli a számlát. Ez a Nagy Probléma! Amíg Monica öntudatlan volt az egyenleg ellenőrzése és a pénz leemelése között, Ryan felébredt, és ő is megterhelte a számlát.

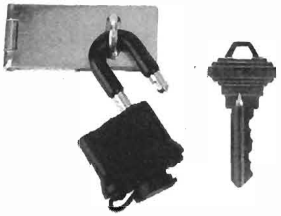
Monica számlaellenőrzése tehát nem volt érvényes, mert Ryan korábban ellenőrizte az egyenleget, és megkezdett egy terhelési műveletet.

Monicát meg kell akadályozni abban, hogy hozzáférhessen a számlához, amíg Ryan fel nem ébred, és be nem fejezi az általa megkezdett tranzakciót (és viszont).

A számla eléréséhez egy zárra van szükség!

A zár így működik:

1. A bankszámla-tranzakcióhoz (az egyenleg ellenőrzéséhez és a pénz leemeléséhez) tartozik egy zár (lock). A zárhoz egyetlen kulcs áll rendelkezésre, ami a zár mellett marad, amíg valaki hozzá nem akar férni a számlához.



Amikor senki nem használja a számlát, a bankszámla-tranzakció zára nyitva van.

2. Amikor Ryan hozzá akar férni a számlához (hogy ellenőrizze az egyenleget, és pénzt emeljen le a számláról), bezárja a zárat, és zsebre vágja a kulcsot, így senki más nem érheti el a számlát, hiszen nincs másik kulcs.



Amikor Ryan hozzá akar férni a számlához, bezárja azt, és magához veszi a kulcsot.

3. **Ryan a zsebében tartja a kulcsot, amíg be nem fejezi a tranzakciót.** Nála van az egyetlen kulcs, így Monica nem férhet hozzá a számlához, amíg Ryan ki nem nyitja a számlát, és vissza nem teszi a kulcsot.

   Így még ha Ryan álomba is merülne, miután ellenőrizte az egyenleget, biztos lehet benne, hogy az egyenleg nem fog megváltozni, amíg alszik, mert nála van a kulcs!



Amikor Ryan végzett, kinyitja a zárat, és visszateszi a kulcsot, így azt Monica (vagy akár ismét Ryan) is elérheti, ha hozzá akar férni a számlához.

A makeWithdrawal() tagfüggvényt atomi műveletté kell tennünk



Gondoskodnunk kell róla, hogy ha egyszer egy szál belépett a makeWithdrawal() tagfüggvénybe, *mindenképpen befejezze azt*, mielőtt egy másik szál közbeszólhatna. Más szavakkal, biztosítanunk kell, hogy ha egy szál ellenőrizte a számlaegyenleget, ez a szál garantáltan felébredhessen, és befejezhesse a terhelést, *mielőtt bármilyen más szál megnézhetné a számlaegyenleget.*

Egy tagfüggvényt a **synchronized** (összehangolt) kulcsszóval módosíthatsz úgy, hogy egyszerre csak egy szál érhesse el.

Így védheted meg a bankszámlát. Nem magára a bankszámlára teszel zárat, hanem azt a tagfüggvényt zárolod, amelyik a banki tranzakciót végzi. Így egy szál az elejétől a végéig végrehajthat egy teljes tranzakciót, még akkor is, ha a tagfüggvény közepén elalszik!

De ha nem a bankszámlára teszünk zárat, akkor pontosan *mit* is zárolunk? A tagfüggvényt? A Runnable objektumot? Magát a szálat?

A következő oldalon meglátod. A kód viszont nagyon egyszerű: csak hozzá kell adnod a synchronized módosítót a tagfüggvény bevezetéséhez:



A `synchronized` kulcsszó azt jelenti, hogy a szálnak egy kulcsra van szüksége ahhoz, hogy hozzáférjen az összehangolt kódhoz.

Ha meg szeretnéd védeni az adataidat (mint a bankszámlát), hangold össze az adatokon műveleteket végző tagfüggvényeket.

```
private synchronized void makeWithdrawal(int amount) {

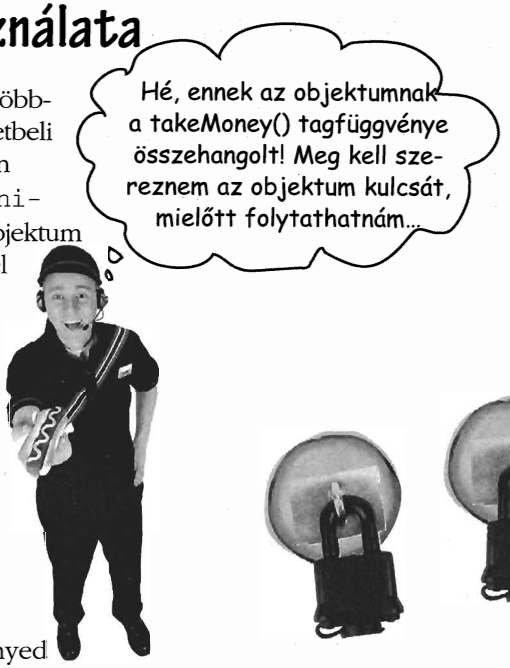
    if (account.getBalance() >= amount) {
       System.out.println(Thread.currentThread().getName() + " is about to withdraw");
       try {
          System.out.println(Thread.currentThread().getName() + " is going to sleep");
          Thread.sleep(500);
       } catch(InterruptedException ex) {ex.printStackTrace(); }
       System.out.println(Thread.currentThread().getName() + " woke up.");
       account.withdraw(amount);
       System.out.println(Thread.currentThread().getName() + " completes the withdrawl");
    } else {
       System.out.println("Sorry, not enough for " + Thread.currentThread().getName());
    }
}
```



(Megjegyzés a fizikarajongóknak: igen, az „atomi" (oszthatatlan) szó itteni használata nem veszi figyelembe a szubatomi részecskéket. Amikor az „atomi" kifejezést szálakkal vagy tranzakciókkal kapcsolatban hallod, Newtonra, és ne Einsteinre gondolj. Hé, ne minket okolj, ezt nem MI találtuk ki! Ha RAJTUNK múlna, lényegében mindenre, ami a szálakkal kapcsolatos, a Heisenberg-féle határozatlansági elvet alkalmaznánk.)

Az objektumzárak használata

Minden objektumnak van zára. A zár a legtöbbször nyitva van, és mellette lóg egy képzeletbeli kulcs. Az objektumzáraknak csak akkor van szerepe, ha a kód összehangolt (`synchronized`) tagfüggvényeket tartalmaz. Ha egy objektum egy vagy több összehangolt tagfüggvénnyel rendelkezik, ***egy szál csak akkor léphet be ezek valamelyikébe, ha meg tudja szerezni az objektum zárának kulcsát.***



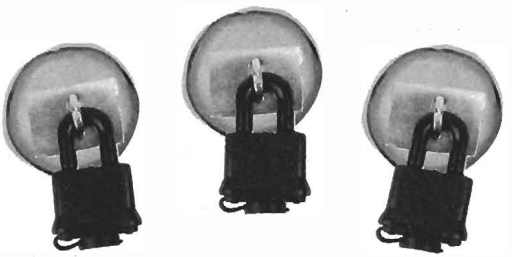
Nincs *tagfüggvényenként* külön zár, csak *objektumonként*. Ha egy objektumnak két összehangolt tagfüggvénye van, az nem csupán annyit jelent, hogy két szál nem léphet be ugyanabba a tagfüggvénybe – a szálak a két összehangolt tagfüggvény *egyikébe* sem léphetnek be.

Gondold végig. Ha több olyan tagfüggvényed is van, amelyek egy objektum példányváltozóin műveleteket hajthatnak végre, e tagfüggvények mindegyikét a synchronized kulcsszóval kell védened.

Az összehangolás célja a létfontosságú adatok védelme, de ne feledd: nem magukat az adatokat zárolod, hanem az adatokhoz *hozzáférő* tagfüggvényeket hangolod össze.

Mi történik, amikor egy szál végighalad a hívási vermén (a run() tagfüggvénnyel kezdve), és hirtelen egy összehangolt tagfüggvénybe botlik? A szál felismeri, hogy az objektumhoz egy kulcsra van szüksége, mielőtt beléphetne a tagfüggvénybe, ezért megkeresi a kulcsot (mindezt a JVM intézi – a Javában nincs API az objektumzárak elérésére), és ha a kulcs hozzáférhető, megkaparintja, és belép a tagfüggvénybe.

Innen kezdve a szál úgy kapaszkodik a kulcsba, mintha az élete függne tőle. A szál nem ereszti el a kulcsot, amíg nem végzett az összehangolt tagfüggvénnyel, így amíg a kezében van a kulcs, más szálak az objektum egyik összehangolt tagfüggvényébe sem léphetnek be, mivel az objektum kulcsához nem férhetnek hozzá.



Minden Java-objektumnak van zára, egy zárhoz pedig csak egyetlen kulcs tartozik.

A zár a legtöbbször nyitva van, és senki nem törődik vele. Ha azonban egy objektum összehangolt tagfüggvényekkel rendelkezik, egy szál CSAK AKKOR léphet be ezek valamelyikébe, ha az objektum zárának kulcsa hozzáférhető – vagyis csak akkor, ha a kulcs még nincs egy másik szál kezében.

Az „Elveszett frissítés" rettegett problémája

Ismerkedj meg egy klasszikus párhuzamossági problémával, ami az adatbázisok világából származik. A példa szorosan összefügg Ryan és Monica történetével, de ezen keresztül néhány további kérdésre is rávilágíthatunk.
Az elveszett frissítés problémája egyetlen eljárás körül forog:
1. lépés: a számlaegyenleg (balance) lekérdezése:

```
int i = balance;
```

2. lépés: az egyenleg növelése 1-gyel:

```
balance = i + 1;
```

A problémát úgy tudjuk bemutatni, hogy a számítógépet arra kényszerítjük, hogy az egyenleg módosítását két lépésben végezze el. A valóságban ezt a műveletet egyetlen utasítással – **`balance++;`** – hajtanánk végre. Azzal azonban, hogy *két* lépést kényszerítünk ki, világossá válik, hogy mi a gond egy ilyen nem atomi művelettel. Most képzeld el, hogy az egyszerű „egyenleg lekérdezése, majd az egyenleg növelése 1-gyel" helyett a két (vagy több) lépés sokkal bonyolultabb, és nem is elég hozzájuk egy-egy utasítás.
Az elveszett frissítés problémájában két szálunk van, és mindkettő növelni próbálja az egyenleget:

```
class TestSync implements Runnable {

    private int balance;

    public void run() {
       for(int i = 0; i < 50; i++) {
        increment();
        System.out.println("balance is " + balance);
      }
    }

    public void increment() {
       int i = balance;
       balance = i + 1;
    }
}

public class TestSyncTest {
   public static void main (String[] args) {
     TestSync job = new TestSync();
     Thread a = new Thread(job);
     Thread b = new Thread(job);
     a.start();
     b.start();
   }
}
```

Mindkét szál 50-szer fut le, minden ismétléskor növelve az egyenleget.

"Az egyenleg: "

Ez a lényeges rész! Az egyenleget úgy növeljük, hogy ahhoz adunk hozzá 1-et, ami a KIOLVASÁSKOR volt a balance értéke (nem pedig ahhoz, ami a JELENLEGI érték).

Futtassuk ezt a kódot...

1. Egy ideig az „A” szál fut



Az egyenleg (balance) értékét az i változóba helyezzük.
Az egyenleg 0, így most az i értéke is 0.
Az egyenleg értékét az i+1 eredményére módosítjuk.
Az egyenleg értéke most 1.
Az egyenleg értékét az i változóba helyezzük.
Az egyenleg 1, így most az i értéke is 1.
Az egyenleg értékét az i+1 eredményére módosítjuk.
Az egyenleg értéke most 2.

2. Egy ideig a „B” szál fut



Az egyenleg értékét az i változóba helyezzük.
Az egyenleg 2, így most az i értéke is 2.
Az egyenleg értékét az i+1 eredményére módosítjuk.
Az egyenleg értéke most 3.
Az egyenleg értékét az i változóba helyezzük.
Az egyenleg 3, így most az i értéke is 3.

[a „B” szálat az ütemező visszaküldi futtatható állapotba, ***mielőtt*** *az egyenleg értékét 4-re állítaná]*

3. Ismét az „A” szál fut, ott folytatva, ahol abbahagyta



Az egyenleg értékét az i változóba helyezzük.
Az egyenleg 3, így most az i értéke is 3.
Az egyenleg értékét az i+1 eredményére módosítjuk.
Az egyenleg értéke most 4.
Az egyenleg értékét az i változóba helyezzük.
Az egyenleg 4, így most az i értéke is 4.
Az egyenleg értékét az i+1 eredményére módosítjuk.
Az egyenleg értéke most 5.

Elvesztettük az érték utolsó frissítését, amit az „A” szál végrehajtott! A „B” szál korábban kiolvasta az egyenleg értékét, és amikor felébredt, úgy folytatta a munkáját, mintha közben mi sem történt volna.

4. Ismét a „B” szál fut, és pontosan ott folytatja, ahol abbahagyta!



Az egyenleg értékét az i+1 eredményére módosítjuk.
Az egyenleg értéke most 4.

Jaj!

Az „A” szál 5-re módosította az értéket, de amikor a „B” szál visszatért, felülírta ezt a módosítást, mintha sohasem történt volna meg.

Tedd az increment() tagfüggvényt atomivá – hangold össze!

Az increment() tagfüggvény összehangolása kiküszöböli az „Elveszett frissítés" problémáját, mert így a tagfüggvény egyetlen, megbonthatatlan egységként kezeli a két lépést.

```
public synchronized void increment() {
    int i = balance;
    balance = i + 1;
}
```

Miután egy szál belépett a tagfüggvénybe, biztosítanunk kell, hogy a tagfüggvény minden lépése befejeződjön (egyetlen atomi műveletként), mielőtt bármely másik szál hozzáférhetne a tagfüggvényhez.

Nincsenek hülye kérdések

K: Ezek szerint célszerű mindent összehangolni a szálbiztosság érdekében, nem?

V: Nem, ez nem jó ötlet. Az összehangolásnak ugyanis ára van. Először is, egy összehangolt tagfüggvény többletterhelést jelent. Más szavakkal, amikor a kód egy összehangolt tagfüggvényhez ér, lelassul (még ha általában nem is venni észre), mert először ki kell derítenie, hogy elérhető-e a kulcs. Másodszor, egy összehangolt tagfüggvény azért is lelassíthatja a programot, mert az összehangolás korlátozza a párhuzamosságot. Másként megfogalmazva, egy összehangolt tagfüggvény arra kényszeríti a többi szálat, hogy álljon sorba, és várja ki, amíg sorra kerül. Ez lehet, hogy a te kódodban nem okoz gondot, de figyelembe kell venned. Harmadszor – és ez a legijesztőbb –, az összehangolt tagfüggvények holtponthoz (deadlock) vezethetnek! (Lásd pár oldallal később.) A követendő alapszabály az, hogy csak azt tedd összehangolttá, amit feltétlenül muszáj. Valójában a tagfüggvényeknél kisebb egységeket is összehangolhatsz; ebben a könyvben ugyan nem csinálunk ilyesmit, de a synchronized kulcsszót egy teljes tagfüggvénynél finomabb szinten, egy vagy több utasítás összehangolására is használhatod.

A doStuff()-ot nem szükséges összehangolni, ezért az egész tagfüggvényt nem is hangoljuk össze.

```
public void go() {
    doStuff();

    synchronized(this) {
        criticalStuff();
        moreCriticalStuff();
    }
}
```

Itt csak ez a két tagfüggvényhívás alkot egyetlen atomi egységet. Ha a synchronized kulcsszót nem egy tagfüggvény bevezetésében, hanem a tagfüggvényen BELÜL használod, meg kell adnod egy argumentumot, ami az az objektum lesz, amelyiknek a kulcsát a szálnak meg kell szereznie.

Bár léteznek más módszerek is, az összehangolást majdnem mindig az aktuális objektumon (this) fogod végrehajtani. Ugyanezt az objektumot zárolnád akkor is, ha a teljes tagfüggvényt összehangolttá tennéd.

① Egy ideig az „A” szál fut



Megpróbálunk belépni az increment() tagfüggvénybe.

A tagfüggvény összehangolt, ezért **meg kell szereznünk a kulcsot** az objektumhoz.
Az egyenleg értékét az i változóba helyezzük.
Az egyenleg 0, így most az i értéke is 0.
Az egyenleg értékét az i+1 eredményére módosítjuk.
Az egyenleg értéke most 1.
Visszatesszük a kulcsot (befejeztük az increment() tagfüggvényt).
Újra belépünk az increment() tagfüggvénybe, és **megszerezzük a kulcsot**.
Az egyenleg értékét az i változóba helyezzük.
Az egyenleg 1, így most az i értéke is 1.
[az „A” szálat az ütemező visszaküldi futtatható állapotba, de mivel nem fejezte be az összehangolt tagfüggvényt, a kulcs az „A” szálnál marad]

② Az ütemező a „B” szálat jelöli ki futásra



Megpróbálunk belépni az increment() tagfüggvénybe. A tagfüggvény összehangolt, ezért meg kell szereznünk a kulcsot.

A kulcs nem elérhető.

[a „B” szál „az objektum kulcsa nem elérhető” váróterembe kerül]

③ Ismét az „A” szál fut, ott folytatva, ahol abbahagyta (ne feledd, a kulcs még mindig nála van)



Az egyenleg értékét az i+1 eredményére módosítjuk.
Az egyenleg értéke most 2.
Visszatesszük a kulcsot.

[az „A” szálat az ütemező visszaküldi futtatható állapotba, de mivel befejezte az increment() tagfüggvényt, a kulcsot NEM tartja magánál]

④ Az ütemező ismét a „B” szálat futtatja



Megpróbálunk belépni az increment() tagfüggvénybe. A tagfüggvény összehangolt, ezért meg kell szereznünk a kulcsot.

A kulcs ezúttal ELÉRHETŐ, így a szál magához veszi.

Az egyenleg értékét az i változóba helyezzük.

[a szál tovább fut…]

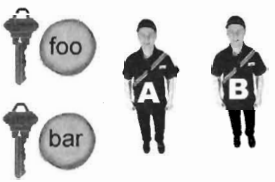
Az összehangolás sötét oldala

Légy óvatos, amikor összehangolt kódot használsz, mert semmi sem tud úgy térdre kényszeríteni egy programot, mint egy szálholtpont. Szálholtpont akkor következik be, amikor két szálad van, és mindkettő egy olyan kulcsot tart magánál, amire a másik szálnak lenne szüksége. Ebből a helyzetből nincs kiút, ezért a két szál csak ül és vár. És vár, és vár, és vár...

Ha vannak tapasztalataid az adatbázisokkal vagy más alkalmazás-kiszolgálókkal kapcsolatban, ismerős lehet a probléma, mert az adatbázisok gyakran alkalmaznak az összehangoláshoz hasonló zárolási módszert. Egy valódi tranzakciókezelő rendszer azonban néha képes megbirkózni a holtpontokkal. Feltételezheti például, hogy a holtpontot az okozta, hogy két tranzakciónak túl sokáig tartott befejeznie a munkáját. A Javával ellentétben azonban egy alkalmazás-kiszolgáló képes „visszagörgetni" a tranzakciókat, vagyis visszaállítani azokat a tranzakció (az atomi művelet) előtti állapotukba.

A Javának ugyanakkor nincs holtpontkezelő rendszere, sőt *tudni* sem fog róla, hogy holtpont következett be, ezért a te feladatod, hogy gondos tervezéssel elkerüld az ilyesmit. Ha sokszor írsz többszálú kódot, érdemes lehet fellapoznod Scott Oaks és Henry Wong *Java Threads* (Java-szálak) című művét, amelyben különféle tervezési tanácsokat találhatsz a holtpontok elkerülésére. Az egyik legfontosabb ezek közül, hogy figyelj oda a szálak indítási sorrendjére.

Egy holtponthoz csak két objektum és két szál kell.



Egyszerű példa egy holtpontra:





Újratöltve

- A statikus Thread.sleep() tagfüggvény arra kényszeríti a szálat, hogy legalább a tagfüggvénynek átadott időtartamra hagyja el a futó állapotot. A Thread.sleep(200) tehát 200 ezredmásodpercre altat el egy szálat.
- A sleep() tagfüggvény ellenőrzött kivételt (InterruptedException) vált ki, ezért minden sleep() hívást egy try/catch blokkba kell burkolni, vagy be kell jelenteni a kivételt.
- A sleep() segítségével biztosíthatod, hogy minden szál esélyt kapjon a futásra, bár arra nincs garancia, hogy amikor egy szál felébred, a futtatható szálak sorának végére kerül. Előfordulhat az is, hogy rögtön a sor elején találja magát. A legtöbb esetben csak megfelelően időzített sleep() hívásokra van szükség ahhoz, hogy a szálak egyenletesen váltogassák egymást.
- Egy szálnak (micsoda meglepetés) a setName() tagfüggvénnyel adhatsz nevet. Minden szál kap egy alapértelmezett nevet, de ha magad adsz nekik nevet, könnyebben nyomon követheted őket, különösen ha print utasításokkal hibakeresést végzel.
- A szálak komoly nehézségeket támaszthatnak, ha két vagy több szálnak ugyanazt az objektumot kell elérnie a kupacban.
- Ha két vagy több szál ugyanazon az objektumon végez műveleteket, az adatok megsérülhetnek, ha mondjuk az egyik szál az objektum valamelyik fontos jellemzőjének módosítása közben hagyja el a futó állapotot.
- Ha szálbiztossá szeretnéd tenni az objektumaidat, döntsd el, hogy mely utasításokat kell egyetlen atomi műveletként kezelni, vagyis hogy mely tagfüggvényeknek kell befejeződniük, mielőtt egy másik szál beléphetne ugyanannak az objektumnak ugyanabba a tagfüggvényébe.
- Ha meg szeretnéd akadályozni, hogy két szál egyidejűleg lépjen be egy tagfüggvénybe, módosítsd a tagfüggvény bevezetését a **synchronized** kulcsszóval.
- Minden objektumhoz egyetlen zár tartozik, amelynek egyetlen kulcsa van. A zárral a legtöbbször nem kell törődnöd; csak akkor van jelentősége, ha az objektum összehangolt tagfüggvényekkel rendelkezik.
- Ha egy szál egy összehangolt tagfüggvénybe szeretne belépni, először meg kell szereznie a tagfüggvényt tartalmazó objektum kulcsát. Amennyiben a kulcs nem elérhető (mert éppen egy másik szál birtokolja), a szál egyfajta „váróterembe" kerül, amíg a kulcs hozzáférhetővé nem válik.
- Akkor is csak egy kulcs van, ha az objektum egynél több összehangolt tagfüggvénnyel rendelkezik. Miután egy szál belépett az objektum valamelyik összehangolt tagfüggvényébe, egyetlen másik szál sem férhet hozzá az említett objektum egyik összehangolt tagfüggvényéhez sem. Ez a korlátozás teszi lehetővé az adataid védelmét a rajtuk műveleteket végző tagfüggvények összehangolásával.

Az új, továbbfejlesztett SimpleChatClient

Még a fejezet elején felépítettünk egy egyszerű csevegőügyfelet (SimpleChatClient), amely üzeneteket tudott *küldeni* a kiszolgálónak, de semmit nem tudott fogadni. Emlékszel? Ezért is kerültek szóba a szálak – szükségünk volt valamilyen módszerre ahhoz, hogy két dolgot csinálhassunk egyszerre: üzeneteket küldeni a kiszolgáló*nak* (a grafikus felületen keresztül), és közben beolvasni és a gördíthető szövegmezőben megjeleníteni a kiszolgáló*ról* érkezőket.

Igen, ennek a fejezetnek is VAN vége, de ez még nem az...

```
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SimpleChatClient {

    JTextArea incoming;
    JTextField outgoing;
    BufferedReader reader;
    PrintWriter writer;
    Socket sock;

    public static void main(String[] args) {
       SimpleChatClient client = new SimpleChatClient();
       client.go();
    }

    public void go() {
```

Ez nagyrészt olyan GUI-kód, amivel már találkoztál. Semmi különös nincs benne, a kiemelt résztől eltekintve, ahol az új, „olvasó" szálat indítjuk el.

```
        JFrame frame = new JFrame("Ludicrously Simple Chat Client");
        JPanel mainPanel = new JPanel();
        incoming = new JTextArea(15,50);
        incoming.setLineWrap(true);
        incoming.setWrapStyleWord(true);
        incoming.setEditable(false);
        JScrollPane qScroller = new JScrollPane(incoming);
        qScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        qScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
        outgoing = new JTextField(20);
        JButton sendButton = new JButton("Send");
        sendButton.addActionListener(new SendButtonListener());
        mainPanel.add(qScroller);
        mainPanel.add(outgoing);
        mainPanel.add(sendButton);
        setUpNetworking();

        Thread readerThread = new Thread(new IncomingReader());
        readerThread.start();

        frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
        frame.setSize(400,500);
        frame.setVisible(true);

    } // a go lezárása
```

Új szálat indítunk egy új belső osztály segítségével, amelyet a szál feladataként (Runnable) használunk. A szál feladata az, hogy olvasson a kiszolgáló csatolófolyamából, és megjelenítse a beérkező üzeneteket a gördíthető szövegmezőben.

```
private void setUpNetworking() {

    try {
       sock = new Socket("127.0.0.1", 5000);
       InputStreamReader streamReader = new InputStreamReader(sock.getInputStream());
       reader = new BufferedReader(streamReader);
       writer = new PrintWriter(sock.getOutputStream());
       System.out.println("networking established");
    } catch(IOException ex) {
       ex.printStackTrace();
    }
  } // a setUpNetworking lezárása
```

A csatoló segítségével jutunk hozzá a bemeneti és kimeneti adatfolyamokhoz. A kimeneti adatfolyamot már használjuk, hogy üzeneteket küldjünk a kiszolgálónak, de most a bemeneti adatfolyamot is használatba vesszük, hogy az új „olvasó" szál olvashasson a kiszolgálóról.

```
public class SendButtonListener implements ActionListener {
   public void actionPerformed(ActionEvent ev) {
      try {
         writer.println(outgoing.getText());
         writer.flush();

      } catch(Exception ex) {
         ex.printStackTrace();
      }
      outgoing.setText("");
      outgoing.requestFocus();
   }
  } // belső osztály lezárása
```

Itt nincs semmi újdonság. Amikor a felhasználó a send (küldés) gombra kattint, ez a tagfüggvény elküldi a szövegmező tartalmát a kiszolgálónak.

```
public class IncomingReader implements Runnable {
   public void run() {
        String message;
        try {

          while ((message = reader.readLine()) != null) {
             System.out.println("read " + message);
             incoming.append(message + "\n");

          } // a while lezárása
        } catch(Exception ex) {ex.printStackTrace();}
     } // a run lezárása
  } // belső osztály lezárása
} // a külső osztály lezárása
```

Ez az, amit a szál csinál!!!

A run() tagfüggvényben egy ciklust ismétel (amíg a kiszolgálótól null-t nem kap), amely egyszerre egy-egy sort olvas be, és hozzáadja azt a gördíthető szövegmezőhöz (egy újsor karakterrel együtt).



Készre sütött kód

A nagyon-nagyon egyszerű csevegőkiszolgáló

Ezt a kiszolgálókódot a csevegőügyfél mindkét változatához használhatod, de minden felelősséget, ami csak felmerülhet, elhárítunk. Ahhoz, hogy a kódot lecsupaszíthassuk, hogy csak azt tartalmazza, ami tényleg elengedhetetlen, kivettünk belőle egy csomó mindent, amire egy valódi kiszolgálóban szükséged lenne. Másként megfogalmazva, a kód működik, de legalább százféleképpen le lehet fagyasztani. Ha a könyv befejezése után szeretnél egy Igazán Jó Hegyezd Ki A Ceruzád gyakorlatot, térj vissza ide, és tedd a kiszolgáló kódját megbízhatóbbá.
Egy másik lehetséges gyakorlat, amit akár rögtön elvégezhetsz, a kód ellátása megjegyzésekkel. Sokkal jobban megértheted a kódot, ha magad próbálsz rájönni, mi történik, mint ha mi magyaráznánk el. Persze ez egy Készre sütött kód, tehát egyáltalán nem muszáj értened. Csak azért van itt, hogy támogassa a csevegőügyfél két változatát.

A csevegőügyfél futtatásához két terminálra lesz szükséged. Először indítsd el ezt a kiszolgálót az egyik terminálban, majd az ügyfelet a másikban.

```
import java.io.*;
import java.net.*;
import java.util.*;

public class VerySimpleChatServer {

    ArrayList clientOutputStreams;

    public class ClientHandler implements Runnable {
         BufferedReader reader;
         Socket sock;

         public ClientHandler(Socket clientSocket) {
            try {
              sock = clientSocket;
              InputStreamReader isReader = new InputStreamReader(sock.getInputStream());
              reader = new BufferedReader(isReader);

            } catch(Exception ex) {ex.printStackTrace();}
          } // a konstruktor lezárása

        public void run() {
            String message;
            try {
              while ((message = reader.readLine()) != null) {
                 System.out.println("read " + message);
                 tellEveryone(message);

              } // a while lezárása
            } catch(Exception ex) {ex.printStackTrace();}
       } // a run lezárása
  } // belső osztály lezárása
```

```
    public static void main (String[] args) {
         new VerySimpleChatServer().go();
    }

    public void go() {
       clientOutputStreams = new ArrayList();
       try {
         ServerSocket serverSock = new ServerSocket(5000);

         while(true) {
            Socket clientSocket = serverSock.accept();
            PrintWriter writer = new PrintWriter(clientSocket.getOutputStream());
            clientOutputStreams.add(writer);

            Thread t = new Thread(new ClientHandler(clientSocket));
            t.start();
            System.out.println("got a connection");
         }

       } catch(Exception ex) {
         ex.printStackTrace();
       }
    }  // a go lezárása

    public void tellEveryone(String message) {

       Iterator it = clientOutputStreams.iterator();
       while(it.hasNext()) {
          try {
             PrintWriter writer = (PrintWriter) it.next();
             writer.println(message);
             writer.flush();
          } catch(Exception ex) {
              ex.printStackTrace();
          }

       }  // a while lezárása

    }  // a tellEveryone lezárása
}  // az osztály lezárása
```

Nincsenek hülye kérdések

K: Mi a helyzet a statikus változók állapotának megőrzésével? Ha olyan statikus tagfüggvényeim vannak, amelyek statikus változók állapotát módosítják, attól még alkalmazhatok összehangolást?

V: Igen! Emlékezz vissza, hogy a statikus tagfüggvényeket egy osztályon, és nem az osztály egy adott példányán futtatjuk. Felmerülhet hát benned a kérdés: egy statikus tagfüggvény melyik objektum zárát használná? Hiszen az is előfordulhat, hogy az adott osztálynak egyáltalán *nincs* példánya. Szerencsére azonban ahogy minden *objektum* rendelkezik saját zárral, úgy minden betöltött *osztály* is. Ez azt jelenti, hogy ha három Kutya objektumod van a kupacban, akkor összesen négy zár kapcsolódik a Kutyához: a három Kutya-példányé, valamint magáé a Kutya osztályé. Amikor egy statikus tagfüggvényt teszel összehangolttá, a Java magának az osztálynak a zárát használja. Ha tehát egy osztályon belül két statikus tagfüggvényt hangolsz össze, a szálaknak az osztály zárát kell megszerezniük, mielőtt a kettő *bármelyikébe* beléphetnének.

K: Mi az a szálprioritás? Úgy hallottam, annak a segítségével szabályozni lehet az ütemezést.

V: A szálprioritás (thread priority) *esetleg* befolyásolhatja az ütemezőt, de erre sincs garancia. A szálprioritás egy számszerű érték, ami közli az ütemezővel (már ha azt érdekli), hogy a szál mennyire fontos a számodra. Az ütemező általában kidobja az alacsonyabb prioritású szálakat a futó állapotból, ha egy magasabb prioritású szál hirtelen futtathatóvá válik, de még egyszer hangsúlyozzuk: erre nincs garancia. Mi azt javasoljuk, hogy csak akkor add meg a prioritást, ha a *teljesítményt* (a sebességet) szeretnéd befolyásolni, de egy program helyességét soha ne alapozd rá.

K: Miért nem hangoljuk össze egyszerűen az osztály összes kiolvasó és beállító tagfüggvényét a megvédeni kívánt adatokkal? Nem lehetett volna például a BankAccount osztály checkBalance() és withdraw() tagfüggvényeit összehangolttá tenni a Runnable osztály makeWithdrawal() tagfüggvényének összehangolása helyett?

V: Az igazság az, hogy valóban ezeket a tagfüggvényeket *kellett volna* összehangolnunk, hogy megakadályozzuk, hogy más szálak valamilyen más módon elérjék őket. Azért nem tettünk így, mert a példában nem volt semmilyen más kód, ami hozzáfért volna a számlához.

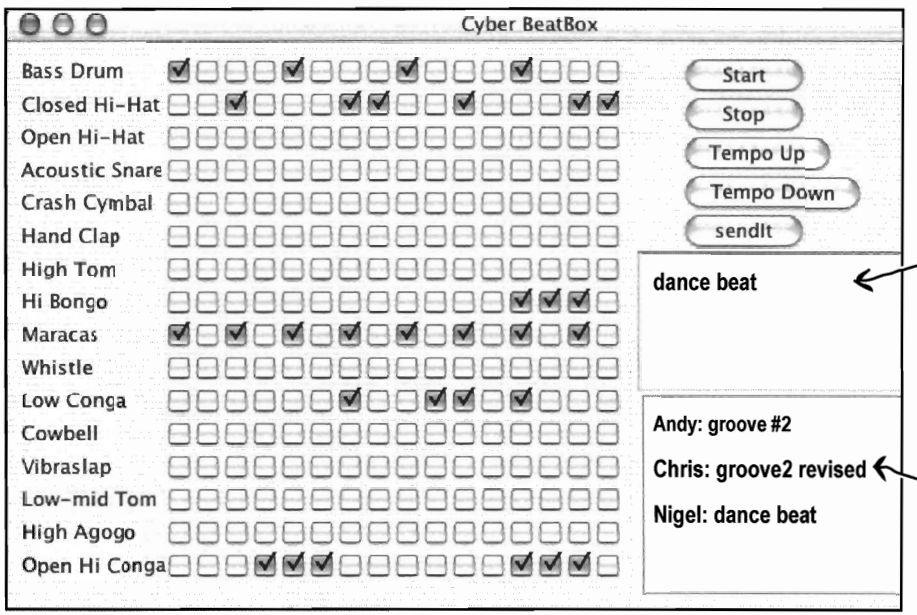
A kiolvasók és beállítók (vagy ebben az esetben a checkBalance() és a withdraw()) összehangolása azonban nem elég. Ne feledd, hogy az összehangolás célja az, hogy egy adott kódrészt ATOMI működésűvé tegyen. Más szavakkal, nem a tagfüggvények az érdekesek, hanem azok a tagfüggvények, ***amelyeknek a befejezéséhez egynél több lépést kell végrehajtani!***

Gondold végig: ha a makeWithdrawal() tagfüggvényt nem hangoltuk volna össze, Ryan megnézte volna az egyenleget (az összehangolt checkBalance() meghívásával), majd rögtön kilépett volna a tagfüggvényből, és visszatette volna a kulcsot!

Természetesen miután felébredt, ismét magához vette volna a kulcsot, hogy meghívhassa a szintén összehangolt withdraw() tagfüggvényt, de ez ugyanazt a problémát eredményezte volna, mint amivel az összehangolás előtt küszködtünk: Ryan ellenőrizheti az egyenleget és elaludhat, majd megérkezhet Monica, és ő is megnézheti az egyenleget, mielőtt Ryan felébredhetne, és befejezhetné a számla megterhelését.

Valószínűleg jó ötlet tehát az összes elérőfüggvényt összehangolni, hogy más szálak ne avatkozhassanak be a műveletbe, azokat a tagfüggvényeket azonban, amelyeknek az utasításait egyetlen, oszthatatlan egységként kell végrehajtani, mindenképpen összehangolttá kell tenni.

Kódkonyha



Az üzenetedet minden felhasználó megkapja az aktuális dobmintáddal együtt, amikor a sendIt gombra kattintasz.

Ezek a többiektől érkező üzenetek. Ha rákattintasz valamelyikre, betöltődik a hozzá kapcsolt minta, és a Start gombra kattintva elindíthatod a lejátszását.

Ez a BeatBox utolsó változata!
Ez a változat egy egyszerű MusicServer kiszolgálóhoz kapcsolódik, hogy dobmintákat cserélhess a többi ügyféllel. A kód nagyon hosszú, ezért teljes egészében csak az „A” függelék mutatja be.





Kódmágnesek

Egy Java-program kódja összekeveredett a hűtőmágneseken. Hozzá tudod adni a következő oldalon szereplő kódrészleteket az alábbi üres osztályokhoz úgy, hogy előálljon belőlük a lap alján látható kimenetet eredményező Java-program? Néhány kapcsos zárójel leesett a padlóra, és túl kicsi volt ahhoz, hogy fel tudjuk venni, úgyhogy nyugodtan egészítsd ki a kódot annyi kapcsos zárójellel, amennyit csak szükségesnek érzel!

```
public class TestThreads {
```

```
class Accum {
```

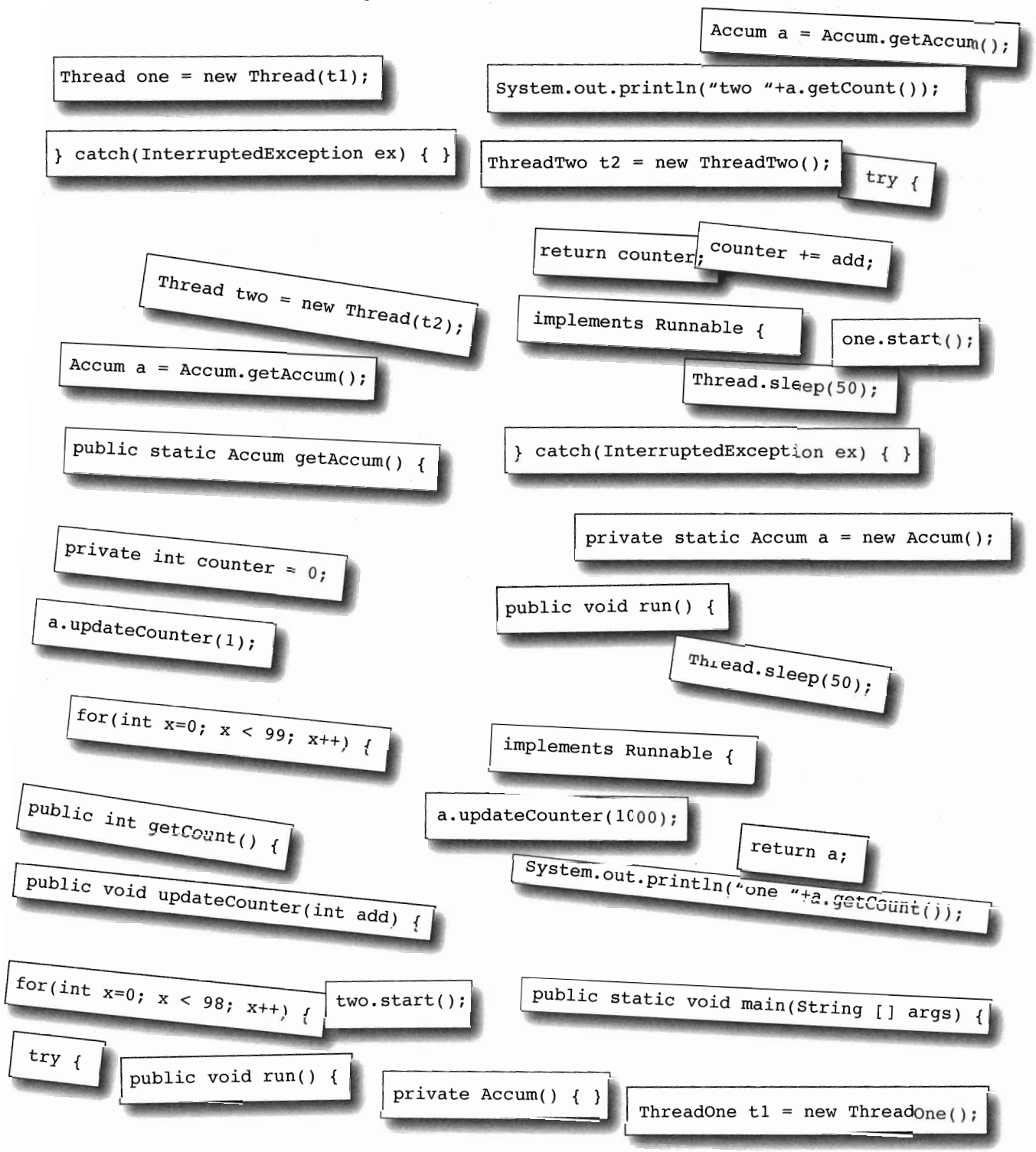
```
class ThreadOne
```

```
class ThreadTwo
```

```
File  Edit  Window  Help  Sewing
% java TestThreads
one 98098
two 98099
```

Bónusz kérdés: Mit gondolsz, miért használtuk az Accum osztály bevezetésében szereplő módosítókat?

Kódmágnesek – folytatás...



Gyakorlat – Megoldás

```
public class TestThreads {
  public static void main(String [] args) {
    ThreadOne t1 = new ThreadOne();
    ThreadTwo t2 = new ThreadTwo();
    Thread one = new Thread(t1);
    Thread two = new Thread(t2);
    one.start();
    two.start();
  }
}

class Accum {
  private static Accum a = new Accum();
  private int counter = 0;

  private Accum() { }

  public static Accum getAccum() {
    return a;
  }

  public void updateCounter(int add) {
    counter += add;
  }

  public int getCount() {
    return counter;
  }
}

class ThreadOne implements Runnable {
  Accum a = Accum.getAccum();
  public void run() {
    for(int x=0; x < 98; x++) {
      a.updateCounter(1000);
      try {
        Thread.sleep(50);
      } catch(InterruptedException ex) { }
    }
    System.out.println("one "+a.getCount());
  }
}
```

statikus példány létrehozása az Accum osztályból

privát konstruktor

Két különböző osztályból származó szál egy harmadik osztálynak ugyanazt az objektumát módosítja, mivel mindkét szál egyetlen Accum-példányon végez műveleteket. A `private static Accum a = new Accum();` kódsor egy statikus Accum-példányt hoz létre (emlékezz vissza, hogy a „statikus" azt jelenti, hogy „osztályonként egy"), az pedig, hogy az Accum konstruktora privát, azt jelenti, hogy senki más nem hozhat létre Accum objektumokat. Ez a két megoldás (a privát konstruktor és a statikus kiolvasófüggvény) együtt alkalmazva az „Egyke" (Singleton) nevű objektumközpontú tervezési mintát eredményezi, amelynek az elsődleges célja az, hogy korlátozza egy objektum példányszámát egy alkalmazásban (egy Egykéből jellemzően egy példány létezik – innen a név), de a minta segítségével tetszőleges módon korlátozhatod a példányok létrehozását.

```
class ThreadTwo implements Runnable {
  Accum a = Accum.getAccum();
  public void run() {
    for(int x=0; x < 99; x++) {
      a.updateCounter(1);
      try {
        Thread.sleep(50);
      } catch(InterruptedException ex) { }
    }
    System.out.println("two "+a.getCount());
  }
}
```



Hajszál híján halálos zsilip

Ötperces rejtély



Ahogy Sarah csatlakozott a fedélzeti fejlesztőcsapathoz, akik tervezési értekezletet tartottak, az ablakon át megcsodálta az Indiai-óceán felett felkelő napot. Bár az űrhajó előadóterme fojtogatóan szűk volt, az egyre növekvő kék-fehér sarló látványa, ahogy átveszi a sötétség helyét az alattuk keringő bolygón, mindig ámulattal töltötte el.
Ezen a reggelen a keringőegység zsilipkapuinak vezérlőrendszere adta az értekezlet témáját. Ahogy az állomás összeszerelése a végéhez közeledett, az előirányzott űrséták száma drámaian megugrott, és mind kifelé, mind befelé jelentős forgalom zajlott a hajó zsilipkapuin át.
– Jó reggelt, Sarah! – köszöntötte Tom – Tökéletes az időzítésed, éppen most akartunk nekikezdeni a terv részletes áttekintésének.
– Amint azt mindnyájan tudjátok, – folytatta Tom – minden zsilipet űrálló grafikus terminállal szereltek fel, kívül-belül egyaránt. Az űrsétára indulók vagy onnan visszaérkezők ezeken a terminálokon keresztül indítják el a zsilipelési eljárást.
– Tom, el tudnád mondani a belépési és kilépési tagfüggvények sorrendjét? – érdeklődött Sarah.
Tom felállt, a táblához lebegett, és gyorsan írni kezdett rá.
– Ez itt a kilépési eljárás álkódja – magyarázta.

```
keringoegysegZsilipKilepesiEljaras()

 ellenorizKapuAllapot();

 levegofeltoltZsilip();

 kinyitBelsoAjto();

 megerositZsilipFoglalt();

 bezarBelsoAjto();

 levegokiszivZsilip();

 kinyitKulsoAjto();

 megerositZsilipUres();

 bezarKulsoAjto();
```

– Annak érdekében, hogy az eljárást ne lehessen megszakítani, – folytatta Tom – a keringoegysegZsilipKilepesiEljaras() által meghívott összes tagfüggvényt összehangolttá tettük. Nem szeretnénk, ha valaki egy űrsétáról visszatérve véletlenül rányitna egy társára, és ott találná letolt nadrággal.
Mindenki nevetett, miközben Tom letörölte a táblát, de Sarah-nak valami szöget ütött a fejébe. Rá is jött, mi zavarja, amint Tom nekilátott, hogy felírja a belépési eljárás álkódját a táblára.
– Várj csak, Tom! – kiáltott fel – Azt hiszem, van egy óriási hiba a kilépési eljárásban! Térjünk vissza hozzá, és vizsgáljuk felül, mert életek múlhatnak rajta!

Miért szakította félbe Sarah az értekezletet? Mire jött rá?



Mire jött rá Sarah?

Sarah arra jött rá, hogy ahhoz, hogy a teljes kilépési eljárás biztosan megszakítás nélkül fusson le, a `keringoegysegZsilipKilepesiEljaras()` tagfüggvényt is összehangolttá kell tenni. A terv jelenlegi állapota szerint előfordulhatott volna, hogy egy űrsétáról visszatérő űrhajós megszakítja a kilépési eljárást. A kilépési eljárást ugyan egyetlen alacsonyabb szintű tagfüggvény futása közben sem lehetett volna megszakítani, az említett hívások *között* azonban igen. Sarah tudta, hogy a teljes utasítássort egyetlen, atomi egységként kell futtatni, és ha a `keringoegysegZsilipKilepesiEljaras()` tagfüggvényt összehangolják, akkor egyetlen ponton sem lesz megszakítható.

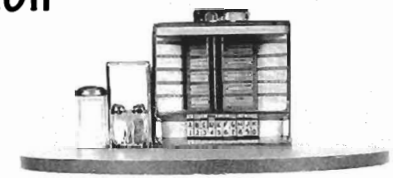
16 Gyűjtemények és általánosítások

Adatszerkezetek



A rendezés pofonegyszerű a Javában. Minden eszköz a kezedben van ahhoz, hogy anélkül módosítsd vagy rendezd gyűjteményekbe az adataidat, hogy saját rendező algoritmusokat kellene írnod (hacsak nem egy alapfokú számítástechnikai órán, a programozás egyszeregyével ismerkedve olvasod ezt, mely esetben – nekünk elhiheted – *biztosan* írni fogsz rendező kódot, amíg mi, többiek, csak meghívjuk a Java API egyik tagfüggvényét). A Java Collections Framework (a Java-gyűjtemények keretrendszere) tartalmaz egy adatszerkezetet, ami lényegében mindenhez jó, amire valaha csak szükséged lehet. Szeretnél egy listát, amelyet bármikor könnyen bővíthetsz? Meg szeretnél találni valamit a neve szerint? Létre szeretnél hozni egy olyan listát, amely önműködően kigyomlálja a többször szereplő elemek ismételt előfordulásait? Sorba akarod állítani a munkatársaidat aszerint, hogy hányszor szúrtak hátba? Vagy a háziállataidat aszerint, hogy hány trükkre tanítottad meg őket? Ezt most mind megtanulhatod...

A dalok népszerűségének nyomon követése egy zenegépen



Gratulálunk az új munkádhoz! Egy automatizált zenegépet (jukebox) kell üzemeltetned Lou kifőzdéjében. Magában a zenegép belsejében nincs Java-program, de amikor valaki kér egy számot, a dal adatai hozzáfűződnek egy egyszerű szövegfájlhoz.

A te feladatod az, hogy az adatok alapján nyomon kövesd a dalok népszerűségét, jelentéseket készíts, és összeállítsd a lejátszási listákat. Nem te írod meg az egész alkalmazást – ebben más szoftverfejlesztők (vagy pincérek) is segítenek, de te felelsz az adatok kezeléséért és rendezéséért a Java-alkalmazásban. Mivel pedig Lou ellenérzésekkel viseltet az adatbázisok iránt, az adatok gyűjteménye szigorúan csak a memóriában létezhet. Te csak azt a fájlt kapod meg, amelynek a tartalmát a zenegép folyamatosan bővíti, és ezt a fájlt kell kiolvasnod.

Azt már kitaláltad, hogyan olvashatod ki és dolgozhatod fel a fájlt, és az adatokat mindezidáig egy tömblistában tároltad:

SongList.txt

```
Pink Moon/Nick Drake
Somersault/Zero 7
Shiva Moon/Prem Joshua
Circles/BT
Deep Channel/Afro Celts
Passenger/Headmix
Listen/Tahiti 80
```

Ebbe a fájlba ír a zenegép. A kódodnak el kell elolvasnia ezt a fájlt, majd műveleteket kell végeznie a daladatokon.

1. feladat

Rendezd a dalokat ábécésorrendbe!

A fájlban dalok listája található, amelyben minden sor egy-egy dalt ír le, a dal címét és előadóját pedig egy perjel választja el. Egy sor feldolgozása és a dalok elhelyezése egy tömblistában tehát elvileg könnyű.

A főnököt csak a számcímek érdeklik, ezért egyelőre elég egy olyan listát készítened, amely csak a dalok címét sorolja fel. A lista azonban láthatóan nem ábécésorrendben van. Mit kezdhetsz vele?

Tudod, hogy a tömblistákban az elemek a beszúrás sorrendjében követik egymást, tehát nem elég egy ArrayListbe tenni őket – valahogy rendezni is kellene az elemeket. Van esetleg egy sort() – „rendez" – tagfüggvény az ArrayList osztályban?

Lássuk, mid van eddig, a rendezés nélkül:

```
import java.util.*;
import java.io.*;

public class Jukebox1 {

    ArrayList<String> songList = new ArrayList<String>();

    public static void main(String[] args) {
        new Jukebox1().go();
    }

    public void go() {
        getSongs();
        System.out.println(songList);
    }

    void getSongs() {
        try {
            File file = new File("SongList.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line = null;
            while ((line= reader.readLine()) != null) {
                addSong(line);
            }

        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/");
        songList.add(tokens[0]);
    }

}
```

A számcímeket egy karakterlánc-tömblistában fogjuk tárolni.

A fájl betöltését elindító és a songList tömblista tartalmát kiíró tagfüggvény.

Itt nincs semmi különös – csak beolvassuk a fájlt, és minden sorra meghívjuk az addSong() tagfüggvényt.

Az addSong() tagfüggvény ugyanúgy működik, mint a QuizCard az I/O-ról szóló fejezetben: a (dal címét és előadóját egyaránt tartalmazó) sort két részre (elemre) bontja a split() tagfüggvény segítségével.

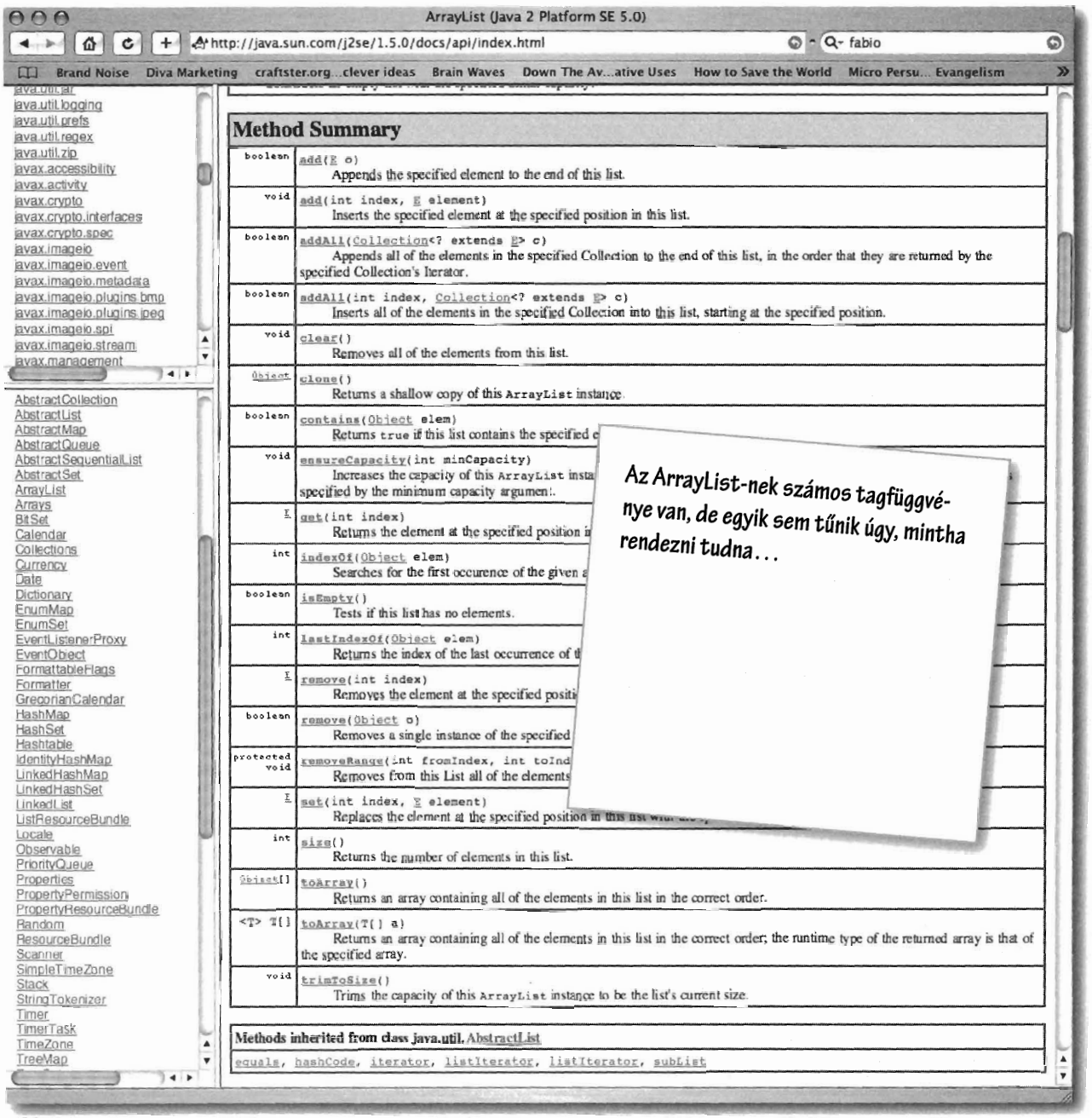
Csak a címre van szükségünk, ezért csak az első elemet adjuk a songList tömblistához.

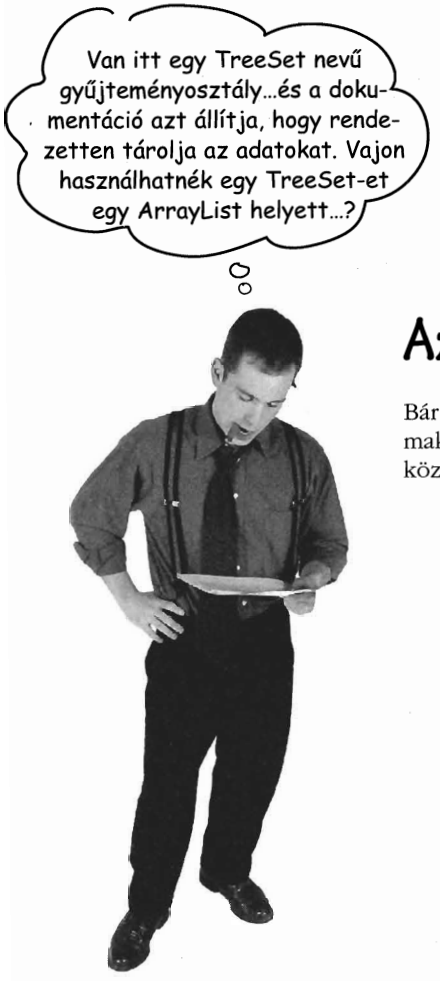
```
File Edit Window Help Dance
%java Jukebox1
[Pink Moon, Somersault,
Shiva Moon, Circles,
Deep Channel, Passenger,
Listen]
```

A songList abban a sorrendben írja ki a címeket, ahogy azokat a tömblistához adtuk (ami megegyezik a dalok sorrendjével az eredeti szövegfájlban). Ez határozottan NEM ábécésorrend!

De az ArrayList osztályban NINCS is sort() tagfüggvény!

Ha megnézed az ArrayList alkalmazásprogramozási felületét, láthatod, hogy egyáltalán nincs benne rendezéssel kapcsolatos tagfüggvény. Az öröklési hierarchiában feljebb lépve sem kapsz segítséget – világos, hogy egy ***ArrayList-re nem hívhatsz meg rendezőfüggvényeket***.





Az ArrayList nem az egyetlen gyűjtemény

Bár az ArrayList-et fogod a leggyakrabban használni, léteznek más, különleges alkalmakra való gyűjtemények is. Lássunk néhányat a legfontosabb gyűjteményosztályok közül:

Ezeknek a megtanulásával most ne vesződj; később részletesebben is szó lesz róluk.

- **TreeSet**
  Az elemeket rendezve tárolja, és megakadályozza az ismétlődéseket.

- **HashMap**
  A segítségével az elemeket név–érték párok formájában tárolhatod és érheted el.

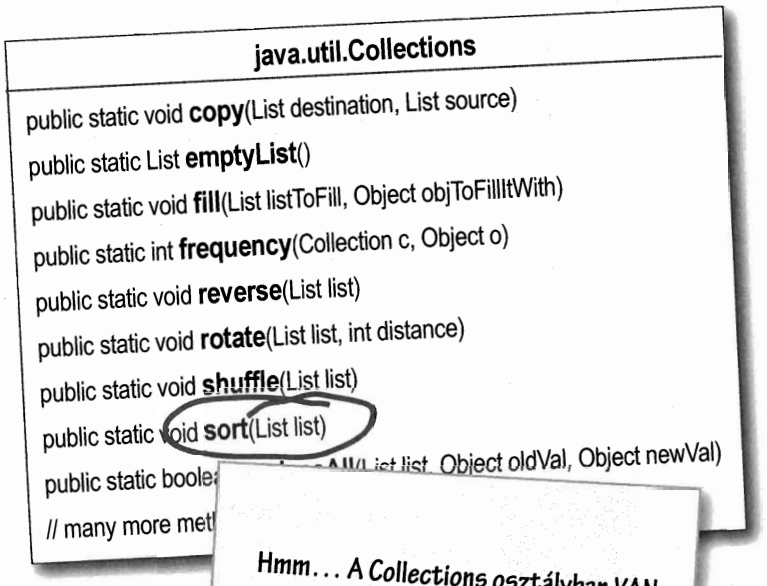
- **LinkedList**
  Arra szánták, hogy felgyorsítsa az elemek beszúrását és törlését a gyűjtemény közepén. (A gyakorlatban általában továbbra is az ArrayList-et fogod használni helyette.)

- **HashSet**
  Megakadályozza az elemek ismétlődését, és a segítségével gyorsan megtalálhatsz egy adott elemet a gyűjteményben.

- **LinkedHashMap**
  Olyan, mint egy szokványos HashMap, azzal a különbséggel, hogy emlékszik az elemek (név–érték párok) beszúrásának sorrendjére, illetve beállítható, hogy emlékezzen rá, hogy az elemeket milyen sorrendben érték el utoljára.

Használhatsz TreeSet-et... de használhatod a Collections.sort() tagfüggvényt is

Ha minden karakterláncot (a dalcímeket) egy **TreeSet**-be („fahalmaz") teszel egy ArrayList helyett, a címek automatikusan a megfelelő helyen landolnak, ábécésorrendben. Akárhányszor íratod ki a listát, az elemei mindig ábécésrorrendben jelennek meg. Ez nagyszerű, ha egy *halmazra* (set; a halmazokról hamarosan részletesebben is szó lesz) van szükséged, vagy ha tudod, hogy a listának mindig ábécésorrendbe rendezve kell maradnia. Más részről azonban, ha a listát nem kell mindig rendezni, a TreeSet költségesebbnek bizonyulhat a szükségesnél, hiszen ***minden alkalommal, amikor beszúrsz valamit a TreeSet-be, a TreeSet-nek időt kell szánnia rá, hogy kideríts, hová kell kerülnie az új elemnek a fában***. Az ArrayList használatakor ezzel szemben a beszúrás villámgyors, mivel az új elem egyszerűen a lista végére kerül.



K: De hát egy ArrayList esetében is LEHET a lista vége helyett egy adott ponton beszúrni elemeket, hiszen létezik egy túlterhelt add() tagfüggvény, amely a beszúrni kívánt elemen kívül egy int értéket vár! Vagy így lassabb beszúrni, mint a lista végére?

V: Igen, egy elemet lassabban lehet beszúrni egy ArrayList *tetszőleges* pontjára, mint a végére. A túlterhelt add(*index, elem*) tagfüggvény tehát nem olyan gyors, mint az add(*elem*), amely az új elemet a lista végére szúrja be. A legtöbbször azonban, amikor ArrayList-et használsz, nem adott helyre kell beszúrnod az elemeket.

K: Látom, van egy LinkedList (láncolt lista) nevű osztály is. Az nem jobb, ha középre akarok beszúrni valamit? Már ha jól emlékszem az adatszerkezetekről szóló órákra a főiskoláról...

V: Jó észrevétel. A LinkedList *valóban* gyorsabb lehet, ha középre kell beszúrnod valamit, vagy középről kell eltávolítanod egy elemet, de a legtöbb alkalmazásban egy LinkedList és egy ArrayList közepére történő beszúrás között nincs akkor eltérés, hogy megérje vele foglalkozni, hacsak nem *óriási* mennyiségű elemről van szó. A LinkedList-re hamarosan bővebben is visszatérünk.

Megjegyzés: A Collections osztály API-ja a valóságban NEM pontosan így fest; az általános típusinformációk (ezekről hamarosan szó lesz) kihagyásával egyszerűsítettünk rajta.

A Collections.sort() hozzáadása a zenegép kódjához

A Collections.sort() tagfüggvény egy karakterláncokból álló listát rendez ábécésorrendbe.

```
import java.util.*;
import java.io.*;

public class Jukebox1 {

    ArrayList<String> songList = new ArrayList<String>();

    public static void main(String[] args) {
        new Jukebox1().go();
    }

    public void go() {
        getSongs();
        System.out.println(songList);
        Collections.sort(songList);
        System.out.println(songList);
    }

    void getSongs() {
        try {
            File file = new File("SongList.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line = null;
            while ((line= reader.readLine()) != null) {
                addSong(line);
            }

        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/");
        songList.add(tokens[0]);
    }
}
```

Meghívjuk a statikus Collections.sort() tagfüggvényt, majd ismét kiíratjuk a listát. A második kiírás már ábécésorrendben lesz!

```
File Edit Window Help Chill
%java Jukebox1

[Pink Moon, Somersault, Shiva Moon, Circles, Deep
Channel, Passenger, Listen]

[Circles, Deep Channel, Listen, Passenger, Pink
Moon, Shiva Moon, Somersault]
```

A sort() meghívása előtt.

A sort() meghívása után.

De így Song objektumokra van szükség, nem egyszerű karakterláncokra!

A főnök egyszercsak meggondolja magát, és valódi Song-osztálypéldányokat akar a listában, nem csupán karakterláncokat, hogy a dalokról több információ álljon rendelkezésre. Az új zenegép több adatot ír ki, ezért ezúttal a fájlban kettő helyett *négy* adatelem fog szerepelni.

A Song osztály nagyon egyszerű. Csak egy érdekes dolog van benne: a felülbírált toString() tagfüggvény. Emlékezz vissza, hogy a toString() meghatározása az Object osztályban található, tehát a Java minden osztálya örökli ezt a tagfüggvényt. Mivel pedig a toString()-et egy objektum kiírásához hívjuk meg (System.out.println(*egyObjektum*)), felül kell bírálnunk, hogy az alapértelmezett egyedi azonosítókód helyett valami értelmesebbet írjon ki. A lista kiíratásakor a toString() tagfüggvényt minden egyes objektumra meghívjuk.

SongListMore.txt

```
Pink Moon/Nick Drake/5/80
Somersault/Zero 7/4/84
Shiva Moon/Prem Joshua/6/120
Circles/BT/5/110
Deep Channel/Afro Celts/4/120
Passenger/Headmix/4/100
Listen/Tahiti 80/5/90
```

Az új fájl kettő helyett négy daljellemzőt tárol. A listában MINDEGYIK jellemzőt szerepeltetni szeretnénk, ezért a Song osztályban mind a négy daljellemzőhöz létre kell hoznunk példányváltozókat.

```
class Song {
    String title;
    String artist;
    String rating;
    String bpm;

    Song(String t, String a, String r, String b) {
        title = t;
        artist = a;
        rating = r;
        bpm = b;
    }

    public String getTitle() {
        return title;
    }

    public String getArtist() {
        return artist;
    }

    public String getRating() {
        return rating;
    }

    public String getBpm() {
        return bpm;
    }

    public String toString() {
        return title;
    }
}
```

Négy példányváltozó a fájlban található négy daljellemzőhöz.

Minden változót a konstruktorban állítunk be, amikor létrehozzuk az új Song objektumot.

A négy jellemzőt kiolvasó tagfüggvények.

Felülírjuk a toString() tagfüggvényt, mert a System.out.println(egySongObjektum) meghívásakor látni szeretnénk a dal címét.
A System.out.println(DalokListája) utasítás a toString() tagfüggvényt a lista MINDEN elemére meghívja.

A Jukebox osztály kódjának módosítása String-ek helyett Song-ok használatára

A kódon csak egy kicsit kell változtatni – a fájl beolvasó–kiíró (I/O) kódja ugyanaz, és a feldolgozó kód (String.split()) sem változik, csak éppen ezúttal minden dal sora *négy* elemet (token) tartalmaz, és mind a négyet felhasználjuk egy-egy új Song objektum létrehozásához. És természetesen az ArrayList most <String> helyett <Song> típusú lesz.

Az ArrayList String-ek helyett Song objektumokból fog állni.

```
import java.util.*;
import java.io.*;

public class Jukebox3 {

    ArrayList<Song> songList = new ArrayList<Song>();
    public static void main(String[] args) {
        new Jukebox3().go();
    }
    public void go() {
        getSongs();
        System.out.println(songList);
        Collections.sort(songList);
        System.out.println(songList);
    }
    void getSongs() {
        try {
            File file = new File("SongList.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line =  null;
            while ((line= reader.readLine()) != null) {
                addSong(line);
            }
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/");

        Song nextSong = new Song(tokens[0], tokens[1], tokens[2], tokens[3]);
        songList.add(nextSong);
    }
```

Létrehozunk egy új Song objektumot a négy adatelemből (vagyis a dalokat tartalmazó fájl adott sorában található négy információból), majd hozzáadjuk a dalt a listához.

A fordítás nem sikerül!

Valami nem stimmel... A Collections osztályban világosan látszik, hogy van egy sort() tagfüggvény, amelynek egy List-et kell átadni. Az ArrayList egyfajta List, mivel megvalósítja a List felületet, tehát a dolognak működnie *kellene*.

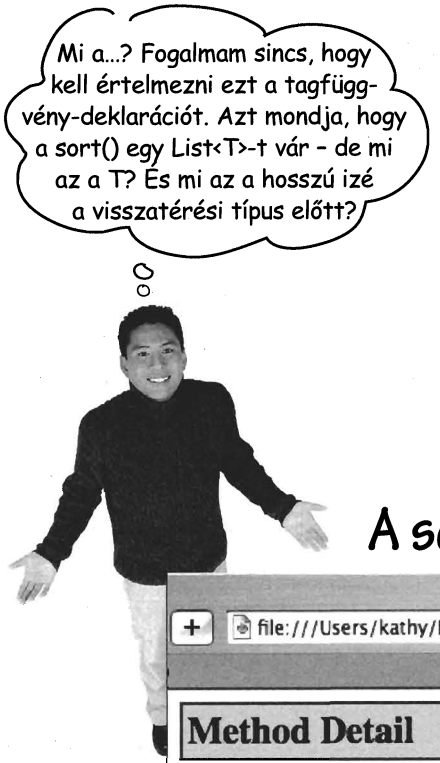
De nem működik!

A fordító azt mondja, hogy nem talál olyan rendező tagfüggvényt, amely egy ArrayList<Song>-ot várna. Lehet, hogy nem tetszik neki a Song objektumokból álló tömblista? Az ArrayList<String>-gel nem volt semmi baja – milyen lényeges eltérés van a Song és a String között? Mi az, ami kudarcra ítéli a fordítót?

File Edit Window Help Bummer

```
%javac Jukebox3.java
Jukebox3.java:15: cannot find symbol
symbol  : method sort(java.util.ArrayList<Song>)
location: class java.util.Collections
                Collections.sort(songList);
                           ^
1 error
```

Természetesen már feltetted magadnak a kérdést: mi *alapján* rendez a tagfüggvény? Honnan *tudhatja* egyáltalán a rendezőfüggvény, hogy mitől „nagyobb" vagy „kisebb" egy dal egy másiknál? Nyilvánvaló, hogy ha azt szeretnéd, hogy a dal *címe* legyen az az érték, amely meghatározza a dalok rendezési sorrendjét, akkor valahogy közölnöd kell a rendezőfüggvénnyel, hogy a címet – és nem mondjuk a percenkénti ütemszámot – kell alapul vennie.
Erre hamarosan sort is kerítünk, de először derítsük ki, miért nem hagyja a fordító, hogy egy Song objektumokból álló tömblistát adjunk át a sort() tagfüggvénynek.



A sort() tagfüggvény deklarációja

Collections (Java 2 Platform SE 5.0)

file:///Users/kathy/Public/docs/api/index.html

Google

Method Detail

sort

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Sorts the specified list into ascending order, according to the *natural ordering* of its elements. All elements in the list must implement the `Comparable` interface. Furthermore, all elements in the list must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

Az API dokumentációjából a java.util.Collections osztályt kikeresve, és a sort() tagfüggvényhez görgetve úgy tűnik, hogy a sort() tagfüggvény deklarációja elég...*furcsa*. De legalábbis semmihez sem hasonlít, amit eddig láttál.

Ennek az az oka, hogy a sort() tagfüggvény (a Java gyűjtemény-keretrendszerének más elemeihez hasonlóan) jelentős mértékben támaszkodik az *általánosításra* (generics). Amikor egy Java-forráskódban vagy -dokumentációban csúcsos zárójeleket látsz, az általánosítást jelent – ez a lehetőség a Java 5.0-ban jelent meg. Úgy fest tehát, hogy előbb meg kell tanulnod értelmezni a dokumentációt, mielőtt kideríthetnéd, miért lehetséges egy String objektumokból álló tömblistát rendezni, egy Song objektumokból állót azonban nem.

Az általánosítás nagyobb típusbiztonságot jelent

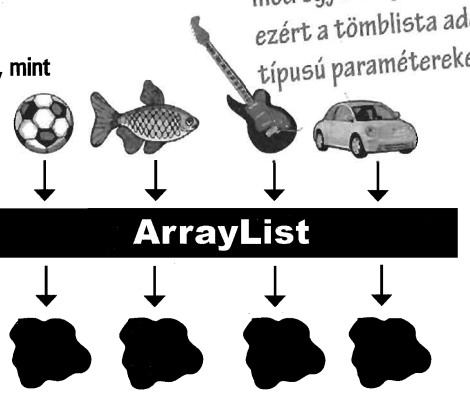
Már most tisztázzuk: *lényegében minden általánosítást tartalmazó kód, amit írsz, gyűjteményekkel lesz kapcsolatos.* Bár az általánosítás más módokon is használható, elsősorban arra való, hogy típusbiztos gyűjteményeket készíts. Más szavakkal, olyan kódot, aminek a hatására a fordítóprogram megakadályozza, hogy egy Kacsákat tartalmazó listába egy Kutyát tegyél.
Az általánosítás megjelenése (vagyis a Java 5.0) előtt a fordítót egyáltalán nem érdekelte, hogy mit teszel egy gyűjteménybe, mivel a deklarációja szerint minden gyűjtemény-megvalósítás Object típusú elemeket tárolt. Egy ArrayList-be a típusától függetlenül *bármit* tehettél; olyan volt, mintha minden ArrayList-et ArrayList<Object>-ként vezettek volna be.

Az általánosításnak köszönhetően típusbiztos gyűjteményeket hozhatsz létre, így sok probléma nem csak futtatáskor, hanem már fordításkor felszínre kerülhet.

Általánosítás nélkül a fordító nem csinált ügyet belőle, ha egy eredetileg csak Macska objektumok tárolására szánt tömblistába egy Tököt tettél.

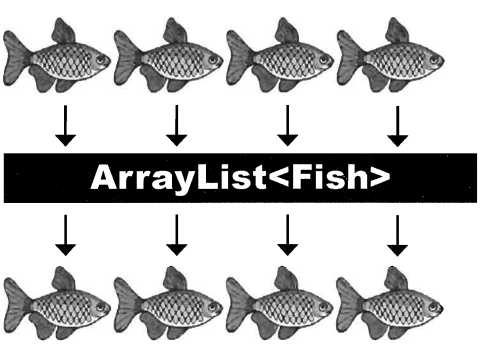
Az általánosítás megjelenése előtt nem volt mód egy ArrayList típusának bevezetésére, ezért a tömblista add() tagfüggvénye Object típusú paramétereket várt.

ÁLTALÁNOSÍTÁS NÉLKÜL
Az objektumok úgy mennek BE, mint Focilabda, Hal, Gitár és Autó objektumokra mutató hivatkozások



És úgy jönnek KI, mint Object típusú hivatkozások

ÁLTALÁNOSÍTÁSSAL
Az objektumok úgy mennek BE, mint Hal objektumokra mutató hivatkozások



És úgy is jönnek KI, mint Hal típusú hivatkozások

Az általánosításnak köszönhetően most már egy ArrayList<Hal>-ba csak Hal objektumokat tehetsz, így az objektumok a kimenetre is Hal hivatkozásokként kerülnek. Nem kell amiatt aggódnod, hogy valaki egy Volkswagent szúr be a listába, vagy hogy a kapott eredményt nem tudod majd Hal hivatkozássá alakítani.

Általánosítás dióhéjban

A több tucat dolog közül, amit az általánosításról megtanulhatsz, valójában csak három számít igazán a legtöbb programozónak:

1. **Az általánosított *osztályok* (mint az ArrayList) példányainak létrehozása**
   Amikor létrehozol egy ArrayList-et, meg kell mondanod neki, hogy milyen típusú objektumokat engedélyezel a listában, ahogy a régi, egyszerű tömbök esetében.

```
new ArrayList<Song>()
```

2. **Általános típusú *változók* bevezetése és értékkel való ellátása**
   Hogyan működik a többalakúság az általános típusok esetében? Egy ArrayList<Animal> típusú hivatkozó változóhoz hozzárendelhetsz egy ArrayList<Dog>-ot is? És mi a helyzet a List<Animal> típusú hivatkozásokkal? Egy ArrayList<Animal> is hozzájuk rendelhető? Mindjárt meglátod...

```
List<Song> songList =
      new ArrayList<Song>()
```

3. **Általános típusú paramétereket váró *tagfüggvények* bevezetése (és meghívása)**
   Mit jelent valójában, ha egy tagfüggvény paraméterként egy Animal objektumokból álló ArrayList-et vár? Egy Dog objektumokat tartalmazó tömblistát is átadhatsz neki? Hamarosan megvizsgálunk néhány trükkös kérdést a többalakúsággal kapcsolatban, és a válaszok igencsak különbözni fognak a régi, egyszerű tömböket váró tagfüggvények írásánál látottaktól.
   (Ez lényegében ugyanaz, mint a 2. pont, de így még jobban fel tudtuk hívni a figyelmedet a kérdés fontosságára.)

```
void foo(List<Song> list)

x.foo(songList)
```

K: Nem kell azt is megtanulnom, hogyan hozhatok létre SAJÁT általános osztályokat? Mi van, ha egy olyan osztálytípust szeretnék létrehozni, amely lehetővé teszi, hogy az osztályt példányosító felhasználó döntse el, hogy milyen típusú dolgokkal működjön az osztály?

V: Erre valószínűleg nem lesz túl gyakran szükséged. Gondold végig: az API tervezői egy teljes könyvtárat építettek fel gyűjteményosztályokból, amelyek a legtöbb adatszerkezetet lefedik, amire csak szükséged lehet, és lényegében a gyűjteményosztályok jelentik az egyetlen olyan osztálytípust, amelynek valóban általánosnak kell lennie. A gyűjteményosztályok ugyanis más elemek tárolására szánt osztályok, és azt szeretnénk, hogy az őket használó programozók határozhassák meg ezeknek az elemeknek a típusát, amikor bevezetnek és példányosítanak egy gyűjteményosztályt. Természetesen előfordulhat, hogy általános osztályt kell *létrehoznod*, de ez inkább a kivételt jelenti, mintsem a szabályt, ezért itt nem foglalkozunk vele. (De ez a könyvben *fellelhető* információkból kikövetkeztethető.)

Általános OSZTÁLYOK használata

Mivel az ArrayList a leggyakrabban használt általános típusunk, az ArrayList leírásának vizsgálatával kezdjük. Egy általános osztályban két kulcsfontosságú dolgot kell megvizsgálnod:
1) Az *osztály* deklarációját
2) Az elemek hozzáadására szolgáló *tagfüggvények* deklarációját

Az ArrayList leírásának értelmezése
(avagy, mit jelent valójában az „E"?)

Az „E"-re gondolj úgy, mint „arra az elemtípusra, amelyet az adott gyűjteményben tárolni és onnan visszaadni szeretnél". (Az „E" az „Elem" rövidítése.)

Az „E" az ArrayList bevezetésekor és létrehozásakor használt VALÓDI típus helyőrzője.

Az ArrayList az AbstractList alosztálya, ezért az AbstractList típusa automatikusan az a típus lesz, amelyet az ArrayList számára megadsz.

```
public class ArrayList<E> extends AbstractList<E> implements List<E>  ... {

      public boolean add(E o)

      // további kód
}
```

Ez a lényeges rész! Az „E" értéke határozza meg, hogy milyen dolgokat lehet hozzáadni az ArrayList-hez.

A típus (az <E> értéke) lesz a List felület típusa is.

Az „E" jelképezi azt a típust, amelyet egy ArrayList-példány létrehozásához használni kell. Ahol „E"-t látsz az ArrayList leírásában, oda azt a típust kell fejben behelyettesítened, amelyet az ArrayList példányosításához használsz. A new ArrayList<Song> tehát azt jelenti, hogy az „E"-ből „Song" lesz minden olyan tagfüggvény- vagy változódeklarációban, ahol „E" szerepelt.

Típusparaméterek használata az ArrayList-tel

EZ a kód...

```
ArrayList<String> thisList = new ArrayList<String>
```

...azt jelenti, hogy az ArrayList-et...

```
public class ArrayList<E> extends AbstractList<E> ... {

      public boolean add(E o)
        // további kód
}
```

...a fordító így kezeli:

```
public class ArrayList<String> extends AbstractList<String>... {

      public boolean add(String o)
        // további kód
}
```

Más szavakkal, az „E" helyére az a *valódi* típus (más néven *típusparaméter*) kerül, amelyet az ArrayList létrehozásakor használsz. Ezért nem enged az ArrayList add() tagfüggvénye semmi mást hozzáadni a listához, csak olyan objektumokat, amelyeknek a hivatkozási típusa összeegyeztethető az „E" típusával. Tehát egy ArrayList**<String>** esetében az add() tagfüggvény hirtelen **add(String o)**-vá válik, míg ha egy **Dog** típusú ArrayList-et hozol létre, az add() tagfüggvényből egyszercsak **add(Dog o)** lesz.

K: Csak „E"-t lehet ide beírni? Mert a sort leírásában „T" szerepelt...

V: Bármit használhatsz, ami érvényes Java-azonosító. Ez azt jelenti, hogy típusparaméterként bármi megfelel, ami egy tagfüggvény vagy változó neveként elfogadható. Hagyományosan azonban egyetlen betűt szoktak használni (ezért te is tégy így), és ez a betű egyezményesen a „T", hacsak nem kifejezetten gyűjteményosztályt írsz, mely esetben az „E" jelképezi „a gyűjteményben tárolni kívánt Elemek típusát".

Általános TAGFÜGGVÉNYEK használata

Az általános *osztályok* olyan osztályok, amelyeknek az *osztálydeklarációja* típusparamétert tartalmaz, az általános *tagfüggvények* pedig olyan tagfüggvények, amelyeknek a deklarációja a tagfüggvény aláírásában alkalmaz típusparamétert.

Egy tagfüggvényben többféleképpen is használhatsz típusparamétereket:

1 Az osztálydeklarációban meghatározott típusparaméter használata

```
public class ArrayList<E> extends AbstractList<E> ... {
    public boolean add(E o)
```

Az „E"-t CSAK AZÉRT használhatod itt, mert már meghatároztad az osztály részeként.

Ha egy osztály számára bevezetsz egy típusparamétert, az adott típust bárhol használhatod, ahol a *tényleges* osztály- vagy felülettípust használnád. A tagfüggvény-argumentumban megadott típus lényegében felcserélődik azzal a típussal, amelyet a osztály példányosításakor használsz.

2 Az osztálydeklarációban NEM meghatározott típusparaméter használata

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

Itt azért használhatod a „T"-t, mert korábban a tagfüggvény-deklarációban bevezetted.

Ha maga az osztály nem használ típusparamétert, akkor is megadhatsz egyet egy tagfüggvény számára, méghozzá igazán szokatlan (de megengedett) helyen bevezetve: a *visszatérési típus előtt*. A fenti tagfüggvény azt mondja, hogy a T bármilyen állatfajta (Animal-altípus) lehet.



Itt jön a bizarr fordulat...

Ez:

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

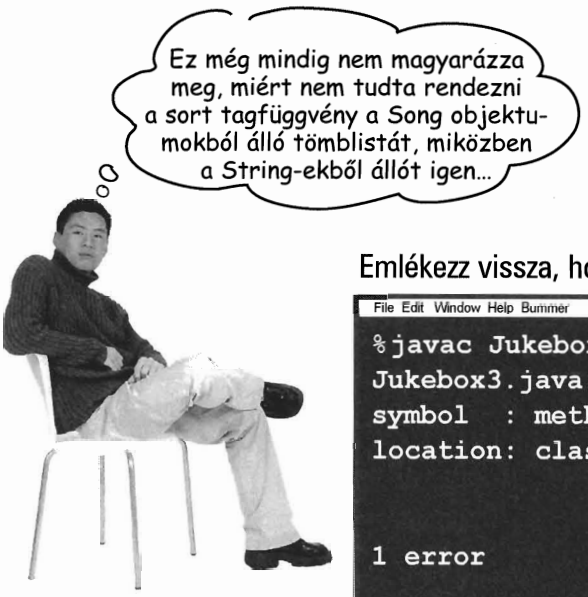
NEM azonos ezzel:

```
public void takeThing(ArrayList<Animal> list)
```

Mindkét alak érvényes, csak éppen *mást* jelent!

Az első típusparaméter – amikor **<T extends Animal>** – a tagfüggvény-deklaráció része, ami azt jelenti, hogy minden olyan ArrayList elfogadható, amelyet Animal típusúként vagy az Animal valamelyik altípusába (például Dog vagy Cat) tartozóként vezettek be. A felső tagfüggvény tehát meghívható ArrayList<Dog>, ArrayList<Cat> és ArrayList<Animal> formában is.
Az alsó alak azonban, ahol a tagfüggvény-argumentum (ArrayList<Animal> list), azt jelenti, hogy *kizárólag* az ArrayList<Animal> elfogadható. Más szavakkal, amíg az első változat bármilyen (Animal, Dog, Cat stb.) Animal típusú tömblistát elfogad, addig a második *csak* Animal típusút: ArrayList<Dog>-ot vagy ArrayList<Cat>-et nem, kizárólag ArrayList<Animal>-t.
Igen, ez valóban úgy tűnik, hogy ellentmond a többalakúság céljának, de majd értelmet nyer, amikor a fejezet végén részletesen átismételjük a tanultakat. Egyelőre ne feledd, hogy most csak azért vetettünk egy pillantást erre a problémára, mert próbáljuk kideríteni, hogy miként rendezzük a dallistánkat, ami elvezetett minket a sort() tagfüggvény API-jában szereplő furcsa általános típusdeklarációhoz.
Egyelőre csak annyit kell tudnod, hogy a felső változat utasításformája érvényes, ami azt jelenti, hogy a tagfüggvénynek egyaránt átadhatsz Animal típusú, illetve bármilyen Animal-altípusúként példányosított ArrayList objektumokat.

És most térjünk vissza a sort() tagfüggvényhez...



Emlékezz vissza, hol tartottunk...

```
File Edit Window Help Bummer
%javac Jukebox3.java
Jukebox3.java:15: cannot find symbol
symbol  : method sort(java.util.ArrayList<Song>)
location: class java.util.Collections
                Collections.sort(songList);
                           ^
1 error
```

```
import java.util.*;
import java.io.*;

public class Jukebox3 {
    ArrayList<Song> songList = new ArrayList<Song>();
    public static void main(String[] args) {
        new Jukebox3().go();
    }
    public void go() {
        getSongs();
        System.out.println(songList);
        Collections.sort(songList);
        System.out.println(songList);
    }
    void getSongs() {
        try {
            File file = new File("SongList.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line =  null;
            while ((line= reader.readLine()) != null) {
                addSong(line);
            }
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/");
        Song nextSong = new Song(tokens[0], tokens[1], tokens[2], tokens[3]);
        songList.add(nextSong);
    }
```

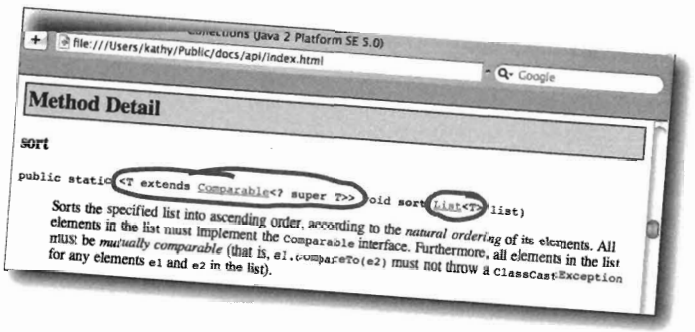
Itt jelentkezett a hiba. A kód rendben működött, amikor egy ArrayList<String>-et adtunk át neki, de kudarcot vallott, amint egy ArrayList<Song>-ot próbáltunk rendezni.

Vissza a sort() tagfüggvényhez

Ott tartottunk tehát, hogy a sort() tagfüggvény leírását vizslatjuk, hogy kiderítsük, miért lehet String-ek listáját rendezni, Song objektumok listáját viszont nem. Úgy tűnik, a válasz a következő:

A sort() tagfüggvény csak Comparable („összehasonlítható") objektumokból álló listákat fogad el.

A Song NEM altípusa a Comparable-nek, ezért egy Song-lista nem rendezhető a sort()-tal. Legalábbis még nem...




```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Ez azt mondja, hogy bármi is legyen a „T", Comparable típusúnak kell lennie.

(Ezzel a résszel egyelőre ne foglalkozz. Ha a kíváncsiság mégis kifúrná az oldalad: csak annyit jelent, hogy a Comparable típusparaméterének típusa T vagy T valamelyik őstípusa kell legyen.)

Csak olyan listát (vagy lista-altípust, például ArrayList-et) adhatsz át, amely a Comparable-t bővítő (extends Comparable) paraméterezett típust használ.



Hm. Megnéztem a String leírását, és a String nem BŐVÍTI (extend) a Comparable-t, hanem MEGVALÓSÍTJA (implements). A Comparable egy felület, ezért hülyeség olyat mondani, hogy <T extends Comparable>.

```
public final class String extends Object implements Serializable,
                                                     Comparable<String>, CharSequence
```

Általánosítás esetén a „bővíti" azt jelenti, hogy „bővíti vagy megvalósítja"

Általánosítás (generics) használata esetén az „extends" (bővíti) kulcsszó valójában azt jelenti, hogy „egyfajta" (is-a), és osztályokra és felületekre EGYARÁNT alkalmazható.

A Java tervezőinek módot kellett adniuk arra, hogy korlátozhasd a paraméterezett típusokat, például ha csak az Animal alosztályait szeretnéd elfogadni. Arra is szükség van azonban, hogy egy típust úgy korlátozz, hogy csak olyan osztályokat fogadj el, amelyek egy adott felületet valósítanak meg. Egy olyan utasításformára volt tehát szükség, ami mindkét helyzetre alkalmazható: öröklésre és megvalósításra, más szavakkal „*extends*" (bővíti) és „*implements*" (megvalósítja) jelentésben is.
A nyertes szó pedig...az *extends* lett. A jelentése azonban igazából „egyfajta" (is-a), és attól függetlenül működik, hogy a jobb oldalon álló típus felület vagy osztály-e.

A Comparable egy felület, ezért ezt VALÓJÁBAN így kell értelmezni: „T-nek olyan típusnak kell lennie, amely megvalósítja a Comparable felületet".

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Nem számít, hogy a jobb oldalon egy osztály vagy egy felület áll, akkor is „extends"-t kell mondanod.

K: Miért nem vezettek be egyszerűen egy új „is" kulcsszót?

V: A nyelvet új kulcsszóval bővíteni NAGYON jelentős változtatás, mert azzal a kockázattal jár, hogy a korábbi Java-változatokban írt kódok működésképtelenné válnak. Gondold végig: lehet, hogy már használsz egy „is" nevű változót (a könyvben a bemeneti adatfolyamok esetében így is tettünk), és mivel a kulcsszavak azonosítóként történő használata tiltott, ez azt jelenti, hogy minden olyan korábbi kód, ami a kulcsszót még az *előtt* használta, hogy az fenntartott szóvá vált volna, működésképtelenné válik. Így aztán amikor csak a Sun tervezői megtehetik, inkább újrahasznosítják a már meglevő kulcsszavakat – ahogy az „extends" esetében is tették. Néha azonban nincs választási lehetőségük...
Néhány (de tényleg csak néhány) új kulcsszó *bekerült* a nyelvbe az idők során, például a Java 1.4-ben az ***assert***, a Java 5.0-ban pedig az ***enum*** (az enum-ról a függelékben ejtünk szót). Ezek a változások valóban működésképtelenné tettek sok kódot, bár néha van rá lehetőség, hogy a Java egy *újabb* változatát és a fordítót úgy futtasd, hogy a viselkedésük a régebbi változatokét utánozza. Ezt úgy érheted el, hogy egy különleges jelzőt adsz át a fordítónak vagy a JVM-nek a parancssorban, ami azt mondja, hogy „Igen, TUDOM, hogy ez a Java 1.4, de kérlek, tégy úgy, mintha az 1.3 lenne, mert a kódomban egy *assert* nevű változót használok, amit még akkor írtam, amikor azt mondtátok, hogy az rendben van!#$%".
(A rendelkezésre álló jelzőket úgy tekintheted meg, hogy beírod a parancssorba a *javac* (a fordító esetében), illetve a *java* (a JVM esetében) parancsot, önmagában, hogy ne álljon utána semmi. Ezekről a jelzőkről a közzétételről szóló fejezetből többet is megtudsz majd.)

Végre tudjuk, mi volt a baj...

A Song osztálynak meg kell valósítania a Comparable felületet

Az ArrayList<Song>-ot csak akkor adhatjuk át a sort() tagfüggvénynek, ha a Song osztály megvalósítja a Comparable felületet, mivel ezt írja elő a sort() tagfüggvény bevezetése. Egy gyors pillantás az API-dokumentációra megmutatja, hogy a Comparable felület igen egyszerű, hiszen csak egyetlen tagfüggvénye van, amelyet meg kell valósítani:

java.lang.Comparable

```
public interface Comparable<T> {
   int compareTo(T o);
}
```

A compareTo() tagfüggvény leírása pedig ezt mondja:

```
Retu
a n
pos
is
tha
   Visszatérési érték:
   negatív egész, nulla vagy pozi-
   tív egész, attól függően, hogy
   az objektum kisebb vagy na-
   gyobb, mint a megadott objek-
   tum, vagy egyenlő azzal
```

Úgy tűnik, a compareTo() tagfüggvény egy Song objektumra fog meghívódni, egy másik Song-ra mutató hivatkozást átadva neki.
A compareTo() tagfüggvényt futtató Song objektumnak meg kell állapítania, hogy a neki átadott Song-ot magasabbra, alacsonyabbra vagy ugyanarra a szintre kell-e helyezni a listában.

A te nagy feladatod most az, hogy eldöntsd, mitől lesz egy dal „nagyobb" egy másiknál, hogy ennek megfelelően valósíthasd meg a compareTo() tagfüggvényt. Egy negatív szám (bármely megatív szám) azt jelenti, hogy az átadott Song „nagyobb", mint a tagfüggvényt futtató Song, míg egy pozitív szám azt, hogy a tagfüggvényt futtató Song nagyobb a compareTo() tagfüggvénynek átadott Song-nál. A nulla visszaadása azt jelenti, hogy a két Song egyenlő (legalábbis a rendezés szempontjából – ez nem feltétlenül jelenti azt, hogy a kettő ugyanaz az objektum): lehet például két dalod, amelyeknek megegyezik a címe. (Ez persze újabb kukacokat hoz a felszínre, de erről majd később...)

A nagy kérdés ez: mitől lesz az *egyik* dal kisebb vagy nagyobb, mint egy *másik* dal, vagy egyenlő vele?

Amíg ezt nem tisztázod, nem tudod megvalósítani a Comparable felületet.



Hegyezd ki a ceruzád!

Írd ide az ötleteidet és az álkódot (vagy ami még jobb, az IGAZI kódot) ahhoz, hogy a compareTo() tagfüggvényt úgy valósíthasd meg, hogy a Song objektumokat cím szerint rendezze.

Tipp: ha a helyes úton jársz, kevesebb, mint 3 sorra lesz szükség!

Az új, javított, összehasonlítható Song osztály

Eldöntöttük, hogy cím szerint akarunk rendezni, ezért a compareTo() tagfüggvényt úgy valósítottuk meg, hogy a tagfüggvénynek átadott dal (Song objektum) címét annak a dalnak a címével hasonlítsa össze, amelyikre a compareTo() tagfüggvényt meghívták. Más szavakkal, a tagfüggvényt futtató dalnak el kell döntenie, hogy a címe hogyan viszonyul a tagfüggvény paraméterében szereplő címhez. Hm... Azt tudjuk, hogy a String osztálynak ismernie kell az ábécésorrendet, hiszen a sort() tagfüggvény karakterláncok listájával működött. Azt is tudjuk, hogy a String-nek van egy compareTo() tagfüggvénye – miért nem hívjuk meg egyszerűen azt? Így hagynánk, hogy az egyik címkarakterlánc (title) összehasonlítsa magát egy másikkal, és nem kellene összehasonlító/ábécésorrendbe állító algoritmust írnunk!

Ezek általában megegyeznek... Azt a típust adjuk meg, amivel a megvalósító osztály összehasonlítható.

Ez azt jelenti, hogy a Song objektumok más Song objektumokkal vethetők össze rendezés céljából.

```
class Song implements Comparable<Song> {
   String title;
   String artist;
   String rating;
   String bpm;

   public int compareTo(Song s) {
       return title.compareTo(s.getTitle());
   }

   Song(String t, String a, String r, String b) {
      title = t;
      artist = a;
      rating = r;
      bpm = b;
   }

   public String getTitle() {
       return title;
   }

   public String getArtist() {
      return artist;
   }

   public String getRating() {
      return rating;
   }

  public String getBpm() {
    return bpm;
  }

  public String toString() {
     return title;
  }
}
```

A sort() tagfüggvény egy Song objektumot ad át a compareTo() tagfüggvénynek, hogy lássa, hogyan viszonyul az átadott Song ahhoz a Song-hoz, amelyikre a tagfüggvényt meghívták.

Egyszerű! Csak átadjuk a munkát a title String objektumoknak, mivel tudjuk, hogy a String-eknek van compareTo() tagfüggvényük.

Most működik. A program kiírja a listát, majd meghívja a rendező-függvényt, ami a dalokat cím szerint ábécérendbe szedi.

File Edit Window Help Ambient

```
%java Jukebox3

[Pink Moon, Somersault, Shiva Moon, Circles, Deep
Channel, Passenger, Listen]

[Circles, Deep Channel, Listen, Passenger, Pink
Moon, Shiva Moon, Somersault]
```

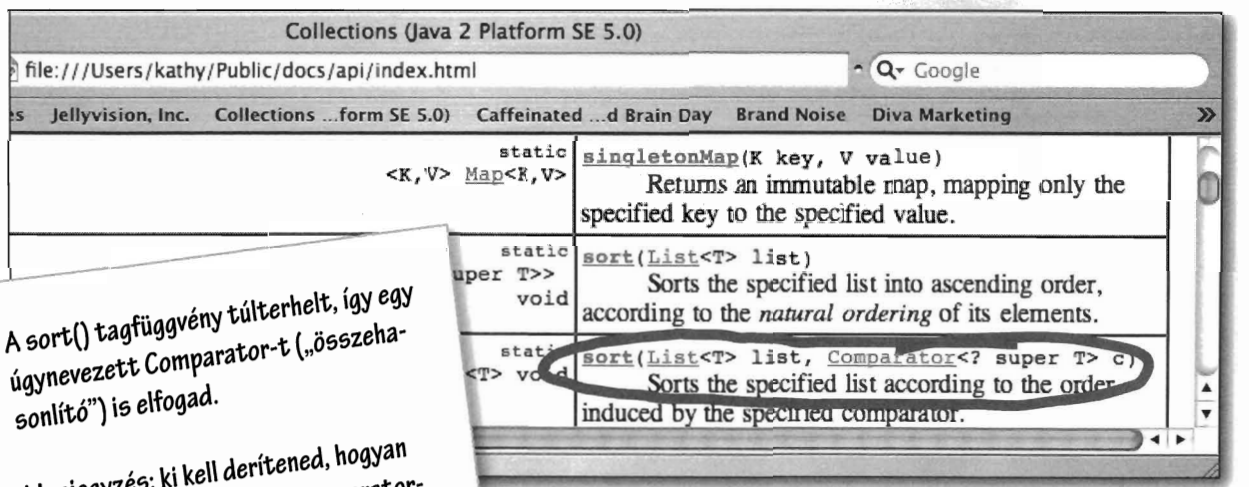
A lista már rendezhető, de...

Van egy kis probléma: Lou két különböző nézetet szeretne a dallistáról – egyet cím, egyet pedig előadó szerint rendezve! Csakhogy amikor egy gyűjteményelemet összehasonlíthatóvá teszel (a Comparable felület megvalósításával), csak egyszer valósíthatod meg a compareTo() tagfüggvényt. Mi lehet a megoldás?

A borzasztó megoldás az lenne, hogy egy jelzőváltozót használsz a Song osztályban, majd a compareTo() tagfüggvényben elvégzel egy *if* tesztet, és attól függően, hogy a jelző a cím vagy az előadó használatára van beállítva, más-más eredményt adnál vissza. Ez azonban esetlen és törékeny megoldás – van jobb, méghozzá olyan, amit kifejezetten erre a célra építettek be az API-ba: arra az esetre, amikor egy dolgot többféleképpen szeretnél rendezni.



Vedd ismét szemügyre a Collections osztály API-ját. Láthatod, hogy van még egy sort() tagfüggvény – és ez egy Comparator-t vár.



A sort() tagfüggvény túlterhelt, így egy úgynevezett Comparator-t („összehasonlító") is elfogad.

Megjegyzés: ki kell derítened, hogyan juthatsz hozzá egy olyan Comparator-hoz, amely képes cím helyett előadó szerint összehasonlítani és rendezni a dalokat...

Egyéni összehasonlító használata

Egy listaelem *önmagát* csak egyféleképpen tudja összehasonlítani egy vele azonos típusú elemmel: a compareTo() tagfüggvénye segítségével. A Comparator („összehasonlító") azonban az összehasonlított elemen kívül található, önálló osztály, vagyis annyi példányt hozhatsz létre belőle, amennyit csak akarsz! Előadó (artist) szerint akarod összehasonlítani a dalokat? Hozz létre egy ArtistComparator-t! Percenkénti ütemszám (bpm, beats per minute) szerint rendeznél? Hozz létre egy BPMComparator-t! Ezután csak annyit kell tenned, hogy meghívod a túlterhelt sort() tagfüggvényt, amely a listát várja, valamint az összehasonlítót, amely majd segít a sort()-nak a rendezésben.
A Comparator-t elfogadó sort() tagfüggvény az elem saját compareTo() tagfüggvénye helyett a Comparator-t használja, amikor az elemeket sorrendbe rakja. Más szavakkal, ha a sort() tagfüggvény egy Comparator-t kap, meg sem *hívja* a lista elemeinek compareTo() tagfüggvényét, hanem a Comparator **compare()** tagfüggfényéhez fordul.
A szabályok tehát a következők:

- **Az egyargumentumos** sort(List o) **tagfüggvény meghívása azt jelenti, hogy a listaelem compareTo() tagfüggvénye határozza meg a sorrendet. A lista elemeinek tehát meg KELL valósítaniuk a Comparable felületet.**
- **A** sort(List o, Comparator c) **meghívása azt jelenti, hogy a listaelem compareTo() tagfüggvényének meghívására NEM kerül sor; ehelyett a Comparator compare() tagfüggvényét használjuk. Ez azt jelenti, hogy a lista elemeinek NEM KELL megvalósítaniuk a Comparable felületet.**

java.util.Comparator

```
public interface Comparator<T> {
   int compare(T o1, T o2);
}
```

Ha sort() tagfüggvénynek átadsz egy Comparator-t, a rendezési sorrendet ez az összehasonlító fogja megszabni, nem pedig az elem saját compareTo() tagfüggvénye.

K: Akkor ez azt jelenti, hogy ha egy osztály nem valósítja meg a Comparable felületet, és nincs meg a forráskódja, egy Comparator létrehozásával azért rendezhetem az elemeket?

V: Így van. A másik lehetőség az lenne (már ha lehetséges), hogy alosztályt származtatsz az elemből, és az alosztályban valósítod meg a Comparable felületet.

K: De miért nem valósítja meg *minden* osztály a Comparable felületet?

V: Tényleg azt hiszed, hogy *minden* rendezhető?

Ha olyan típusú elemeid vannak, amelyek nem rendezhetők semmilyen természetes módon, csak félrevezetnéd a programozókat a Comparable megvalósításával. A meg nem valósítással azonban nem vállalsz különösebb kockázatot, hiszen egy programozó bármit tetszőleges módon összehasonlíthat egy saját, egyéni Comparator segítségével.

A zenegép átállítása összehasonlító használatára

Az alábbi kódban három dolgot teszünk:

1) Létrehozunk egy belső osztályt, amely megvalósítja a Comparator-t (és így a ***compare()*** tagfüggvényt, amely elvégzi a korábban a ***compareTo()*** által elvégzett feladatot).
2) Létrehozzuk a Comparator belső osztály egy példányát.
3) Meghívjuk a túlterhelt sort() tagfüggvényt, és átadjuk neki mind a dallistát, mind a Comparator belső osztály példányát.
   Megjegyzés: a Song osztály toString() tagfüggvényét is frissítjük, hogy a dal címe mellett az előadóját is kiírja. (A kiírás a ***cím: előadó*** formát követi, a lista rendezési módjától függetlenül.)

```
import java.util.*;
import java.io.*;

public class Jukebox5 {
   ArrayList<Song> songList = new ArrayList<Song>();
   public static void main(String[] args) {
      new Jukebox5().go();
   }

   class ArtistCompare implements Comparator<Song> {
     public int compare(Song one, Song two) {
         return one.getArtist().compareTo(two.getArtist());
     }
   }

   public void go() {
     getSongs();
     System.out.println(songList);
     Collections.sort(songList);
     System.out.println(songList);

     ArtistCompare artistCompare = new ArtistCompare();
     Collections.sort(songList, artistCompare);

     System.out.println(songList);
   }

   void getSongs() {
       // ide jön az I/O-kód
   }

   void addSong(String lineToParse) {
       // sor feldolgozása és
       // hozzáadása a dallistához
   }
```

Létrehozunk egy új belső osztályt, amely megvalósítja a Comparator-t. (Vedd észre, hogy a típusparamétere megegyezik az összehasonlítani kívánt elemek – esetünkben a Song objektumok – típusával.)

Ebből egy karakterlánc (az előadó) lesz.

Hagyjuk, hogy a karakterlánc-változók (előadó) végezzék el a tényleges összehasonlítást, mivel a String-ek tudják, hogyan állítsák magukat ábécésorrendbe.

Meghívjuk a sort()-ot, átadva neki a listát és egy hivatkozást az új, egyéni Comparator objektumra.

Megjegyzés: Azzal, hogy a Song compareTo() tagfüggvényében meghagytuk a címek használatát, a cím szerinti rendezést alapértelmezetté tettük, de úgy is megtervezhettük volna a zenegépet, hogy a cím szerinti és az előadó szerinti rendezést is belső Comparator osztályokként valósítjuk meg, és a Song-gal egyáltalán nem valósíttatjuk meg a Comparable felületet. Ilyen módon mindig a Collections.sort() kétargumentumos változatát használnánk.

```
import _____________________;

public class SortMountains {

  LinkedList_____________ mtn = new LinkedList_____________();

  class NameCompare _____________________________________ {
    public int compare(Mountain one, Mountain two) {
      return _____________________________;
    }
  }
  class HeightCompare _____________________________ {
    public int compare(Mountain one, Mountain two) {
      return (_____________________________);
    }
  }
  public static void main(String [] args) {
    new SortMountain().go();
  }
  public void go() {
    mtn.add(new Mountain("Longs", 14255));
    mtn.add(new Mountain("Elbert", 14433));
    mtn.add(new Mountain("Maroon", 14156));
    mtn.add(new Mountain("Castle", 14265));
    System.out.println("as entered:\n" + mtn);
    NameCompare nc = new NameCompare();
    _____________________________________;
    System.out.println("by name:\n" + mtn);
    HeightCompare hc = new HeightCompare();
    _____________________________________;
    System.out.println("by height:\n" + mtn);
  }
}

class Mountain {

  ___________________;
  ___________________;

  _____________________ {
    ___________________;
    ___________________;
  }
  _____________________ {
    _____________________________________;
  }
}
```



Fordított programozás

Tegyük fel, hogy ez a kód egyetlen fájlban található. A feladatod az, hogy kitöltsd az üres helyeket, úgy, hogy a program a lap alján látható kimenetet eredményezze.

Megjegyzés: a megoldás a fejezet végén található.

Kimenet:

```
File  Edit  Window  Help  ThisOne'sForBob
%java SortMountains
as entered:
[Longs 14255, Elbert 14433, Maroon 14156, Castle 14265]
by name:
[Castle 14265, Elbert 14433, Longs 14255, Maroon 14156]
by height:
[Elbert 14433, Castle 14265, Longs 14255, Maroon 14156]
```



Töltsd ki az üres helyeket!

Töltsd ki az alábbi kérdések melletti üres helyeket a „Lehetséges válaszok" listában található szavak felhasználásával. A helyes válaszokat a fejezet végén találod.

Lehetséges válaszok:

Comparator
Comparable
compareTo()
compare()
igen
nem

Ha adott a következő lefordítható utasítás:

```
Collections.sort(myArrayList);
```

1. Mit kell a `myArrayList`-ben tárolt objektumok osztályának megvalósítania?
2. Melyik tagfüggvényt kell a `myArrayList`-ben tárolt objektumok osztályának megvalósítania?
3. A `myArrayList`-ben tárolt objektumok osztálya megvalósíthatja a `Comparator` ÉS a `Comparable` felületet is?

Ha adott a következő lefordítható utasítás:

```
Collections.sort(myArrayList, myCompare);
```

4. A `myArrayList`-ben tárolt objektumok osztálya megvalósíthatja a `Comparable` felületet?
5. A `myArrayList`-ben tárolt objektumok osztálya megvalósíthatja a `Comparator`-t?
6. A `myArrayList`-ben tárolt objektumok osztályának meg kell valósítania a `Comparable` felületet?
7. A `myArrayList`-ben tárolt objektumok osztályának meg kell valósítania a `Comparator`-t?
8. Mit kell megvalósítania a `myCompare` objektum osztályának?
9. Melyik tagfüggvényt kell megvalósítania a `myCompare` objektum osztályának?

Ó, jaj! A rendezés működik, de az elemek ismétlődnek...

A rendezés nagyszerűen működik; most már tudjuk, hogyan rendezhetünk *cím* szerint (a Song objektum compareTo() tagfüggvényét használva) és *előadó* szerint is (a Comparator compare() tagfüggvényével). Viszont felmerül egy új probléma, amit nem vettünk észre a zenegép szövegfájl-mintájában: a ***rendezett listában az elemek ismétlődnek***.
Úgy tűnik, hogy a kifőzde zenegépe mindig hozzáírja a számot a fájlhoz, függetlenül attól, hogy az adott dalt korábban már lejátszották-e (és így beleírták a szövegfájlba). A SongListMore.txt fájl minden lejátszott dalt minden alkalommal rögzít, így ugyanaz a dal többször is szerepelhet benne.

```
File Edit Window Help TooManyNotes
%java Jukebox4

[Pink Moon: Nick Drake, Somersault: Zero 7, Shiva Moon: Prem
Joshua, Circles: BT, Deep Channel: Afro Celts, Passenger:
Headmix, Listen: Tahiti 80, Listen: Tahiti 80, Listen: Tahiti
80, Circles: BT]

[Circles: BT, Circles: BT, Deep Channel: Afro Celts, Listen:
Tahiti 80, Listen: Tahiti 80, Listen: Tahiti 80, Passenger:
Headmix, Pink Moon: Nick Drake, Shiva Moon: Prem Joshua,
Somersault: Zero 7]

[Deep Channel: Afro Celts, Circles: BT, Circles: BT, Passenger:
Headmix, Pink Moon: Nick Drake, Shiva Moon: Prem Joshua, Listen:
Tahiti 80, Listen: Tahiti 80, Listen: Tahiti 80, Somersault:
Zero 7]
```

A rendezés előtt

A Song saját compareTo() tagfüggvényével való (cím szerinti) rendezés után

Az ArtistCompare Comparator segítségével elvégzett (előadó szerinti) rendezés után

SongListMore.txt

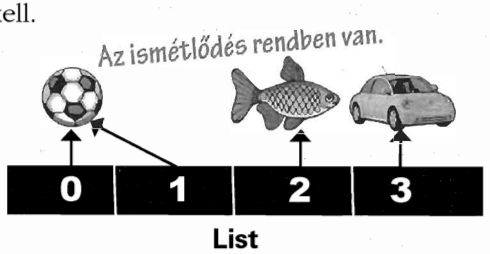
```
Pink Moon/Nick Drake/5/80
Somersault/Zero 7/4/84
Shiva Moon/Prem Joshua/6/120
Circles/BT/5/110
Deep Channel/Afro Celts/4/120
Passenger/Headmix/4/100
Listen/Tahiti 80/5/90
Listen/Tahiti 80/5/90
Listen/Tahiti 80/5/90
Circles/BT/5/110
```

A SongListMore szövegfájlban az elemek ismétlődnek, mert a zenegép sorrendben minden lejátszást rögzít. Valaki háromszor kérte a „Listen"-t, aztán a „Circles"-t, ami már korábban is szerepelt.
A szövegfájl írási módját nem tudjuk megváltoztatni, mert előfordulhat, hogy minden benne levő információra szükségünk lesz. Tehát a Java-kódot kell módosítanunk.

Lista helyett halmazra van szükség

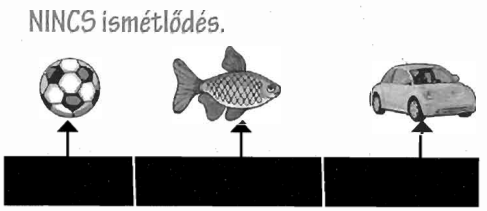
A Collection API-ban három fontos felület található: a **List** (lista), a **Set** (halmaz) és a **Map** („térkép", leképezés, megfeleltetés). Az ArrayList egy **lista**, de úgy fest, nekünk éppen egy **halmaz** kell.

- **LIST – amikor a *sorrend* a lényeg**
  A listák olyan gyűjtemények, amelyek ismerik az elemek ***helyét*** („indexpozícióját").
  A listák tehát tudják, hogy egy elem hol található a gyűjteményben. Ugyanarra az objektumra több elem is hivatkozhat.



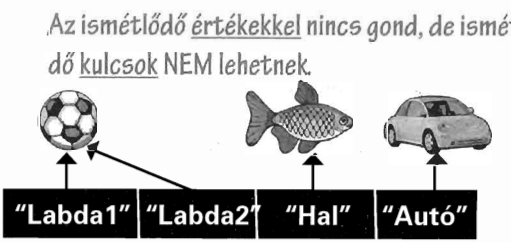
List

- **SET – amikor az *egyediség* a lényeg**
  A halmazok olyan gyűjtemények, amelyek ***nem engedik meg az ismétlődést***.
  A halmazok tudják, hogy egy elem szerepel-e már a gyűjteményben. Ugyanarra az objektumra soha nem hivatkozhat több elem (és két egyenlőnek tekintett objektumra sem – az objektumok egyenlőségéről/azonosságáról hamarosan szót ejtünk).



Set

- **MAP – amikor a *kulcs szerinti keresés* a lényeg**
  A térképek olyan gyűjtemények, amelyek ***kulcs–érték párokat*** használnak.
  A térképek tudják, hogy milyen érték társul egy adott kulcshoz. Ugyanarra az értékre több kulcs is hivatkozhat, de ismétlődő kulcsok nem lehetnek. Bár a kulcsok jellemzően karakterlánc formátumú nevek (így létrehozhatsz például név–érték alakú tulajdonságlistákat), kulcs bármilyen objektum lehet.

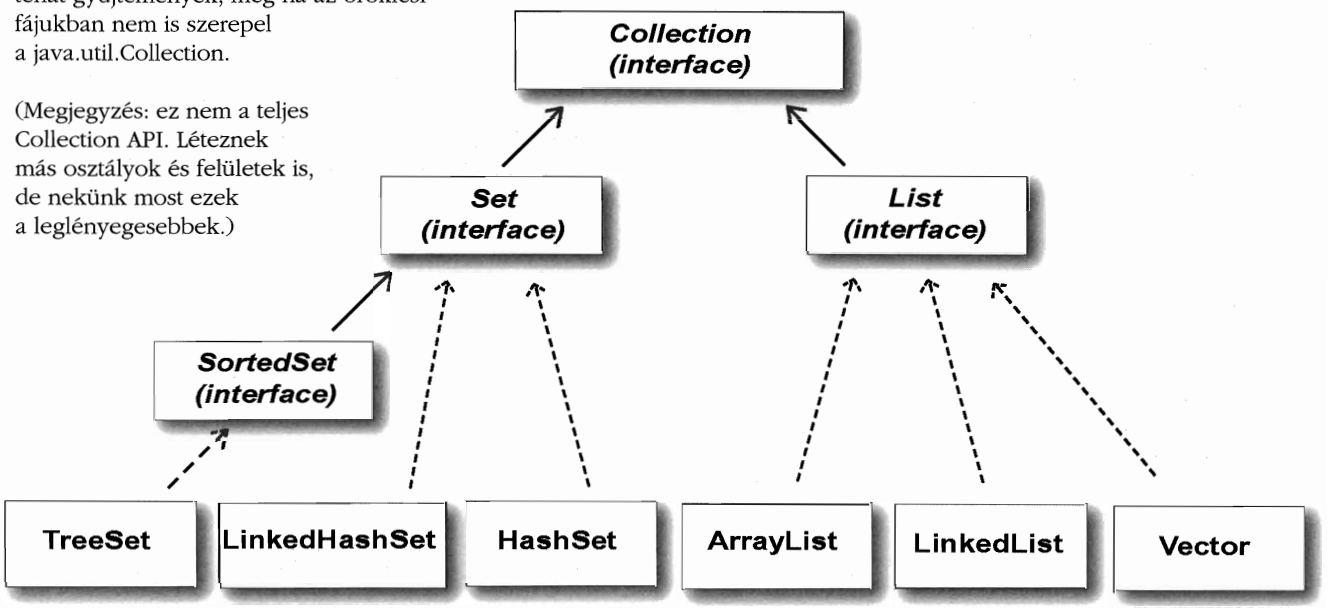


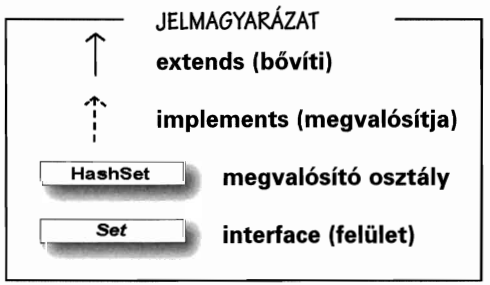
Map

A Collection API (egy része)

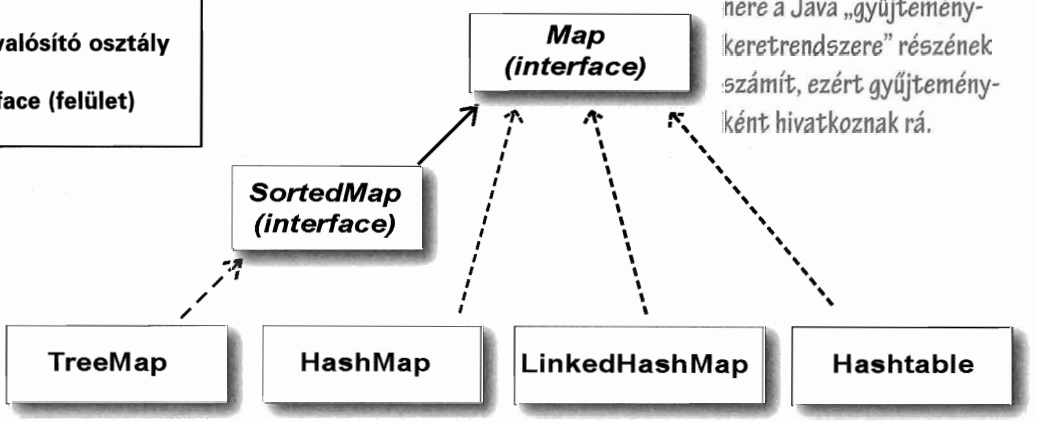
Figyeld meg, hogy a Map felület valójában nem bővítése a Collection felületnek, mégis úgy tekintenek rá, mint „a gyűjtemények keretrendszerének" (Collection Framework, más néven Collection API) részére. A térképek tehát gyűjtemények, még ha az öröklési fájukban nem is szerepel a java.util.Collection.

(Megjegyzés: ez nem a teljes Collection API. Léteznek más osztályok és felületek is, de nekünk most ezek a leglényegesebbek.)





A Map nem leszármazottja a java.util.Collection osztálynak, de ennek ellenére a Java „gyűjtemény-keretrendszere" részének számít, ezért gyűjteményként hivatkoznak rá.



HashSet használata ArrayList helyett

Úgy döntöttünk, hogy a zenegéppel egy hasítóhalmazba (HashSet) íratjuk a dalokat. (Megjegyzés: a Jukebox osztály kódjának egy részét elhagytuk, de kimásolhatod a korábbi változatokból. Ezenkívül a kimenet olvasásának megkönnyítése végett visszatértünk a Song toString() tagfüggvényének korábbi változatához, hogy csak a címet írja ki a cím *és* az előadó helyett.)

```
import java.util.*;
import java.io.*;

public class Jukebox6 {
    ArrayList<Song> songList = new ArrayList<Song>();
    // fő tagfüggvény stb.

    public void go() {
        getSongs();
        System.out.println(songList);
        Collections.sort(songList);
        System.out.println(songList);
        HashSet<Song> songSet = new HashSet<Song>();
        songSet.addAll(songList);
        System.out.println(songSet);
    }
    // a getSongs() és addSong() tagfüggvények
}
```

A getSongs() tagfüggvényen nem változtattunk, ezért az továbbra is egy tömblistába helyezi a dalokat.

Itt hozunk létre egy új HashSet-et, amelyet úgy paraméterezünk, hogy Song objektumokat tároljon.

A HashSet-nek van egy egyszerű addAll() tagfüggvénye, amelynek átadhatunk egy másik gyűjteményt, amellyel feltöltjük a hasítóhalmazt. Ez ugyanolyan, mint ha minden dalt egyenként adnánk hozzá (csak éppen sokkal egyszerűbb).

```
File Edit Window Help GetBetterMusic
%java Jukebox6

[Pink Moon, Somersault, Shiva Moon, Circles, Deep Channel,
Passenger, Listen, Listen, Listen, Circles]

[Circles, Circles, Deep Channel, Listen, Listen, Listen,
Passenger, Pink Moon, Shiva Moon, Somersault]

[Pink Moon, Listen, Shiva Moon, Circles, Listen, Deep Channel,
Passenger, Circles, Listen, Somersault]
```

Az ArrayList rendezése előtt

Az ArrayList (cím szerinti) rendezése után

A halmaz nem segített!!!
Még mindig megvan minden ismétlődés!

(És a rendezési sorrendet elveszítettük, amikor a listát beletettük a hasítóhalmazba, de emiatt most ne aggódjunk...)

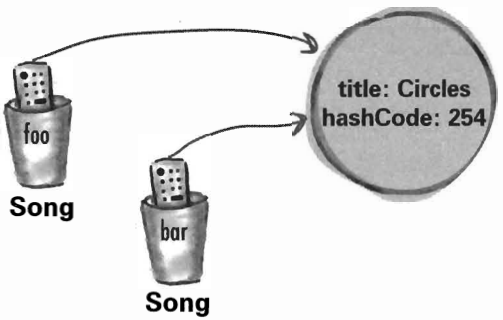
A hasítóhalmazba helyezés és a halmaz tartalmának kiírása után (a sort()-ot nem hívtuk meg újra).

Mi tesz két objektumot azonossá?

Ha a *foo* és a *bar* objektum egyenlő (azonos), akkor a *foo.equals(bar)* állításnak igaznak kell lennie, és a *foo* és a *bar* ugyanazt az értéket kell visszaadja a *hashCode()* hívásból. Ahhoz, hogy egy Set két objektumot ismétlődőként („másolatként", duplicate) kezeljen, felül kell bírálnod az Object osztálytól örökölt hashCode() és equals() tagfüggvényeket, hogy két különböző objektumot azonosnak tekinthessenek.

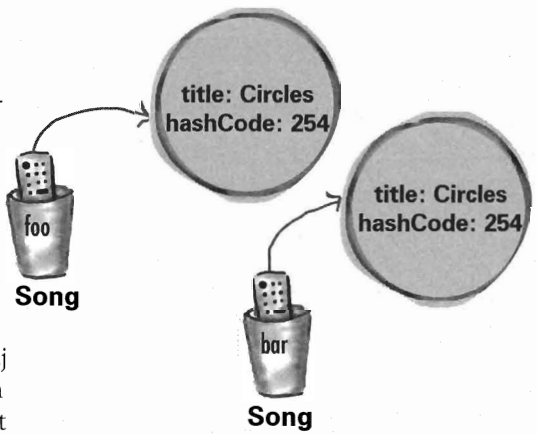
Először azt a kérdést kell feltennünk, hogy mitől számít két Song hivatkozás ismétlődőnek? Attól, hogy ***egyenlőnek*** vagy ***azonosnak*** számítanak. Ez egyszerűen annyit tesz, hogy a két hivatkozás ugyanarra az objektumra mutat, vagy két külön objektumról van szó, amelyeknek a *title* (cím) jellemzője azonos? A kulcsfontosságú kérdés, amit tisztáznunk kell, tehát az, hogy *hivatkozási azonosságról* vagy *objektumazonosságról* van szó?

- Hivatkozási **azonosság**
  Két hivatkozás, egy objektum a kupacban.
  Két hivatkozás azonosnak számít, ha ugyanarra az objektumra mutat a kupacban. Pont. Ha mindkét hivtkozásra meghívod a **hashCode()** tagfüggvényt, ugyanazt az eredményt fogod kapni. Ha nem bírálod felül a hashCode() tagfüggvényt (ne feledd, hogy ezt a tagfüggvényt az Object osztálytól örökölted), az alapértelmezett viselkedés az, hogy minden objektum egyedi azonosítószámot kap. (A Java legtöbb változata a kupacban elfoglalt memóriacímük alapján rendel hasítókódot az objektumokhoz, ezért két objektumnak soha nem lesz ugyanaz a hasítókódja). Ha tudni szeretnéd, hogy két *hivatkozás* valóban ugyanarra az objektumra mutat-e, használd az == műveletet, ami – ha még emlékszel rá – a változókban tárolt biteket hasonlítja össze. Ha mindkét hivatkozás ugyanarra az objektumra mutat, a bitek azonosak lesznek.




```
if (foo == bar) {
   // mindkét hivatkozás ugyanarra
   // az objektumra mutat a kupacban
}
```

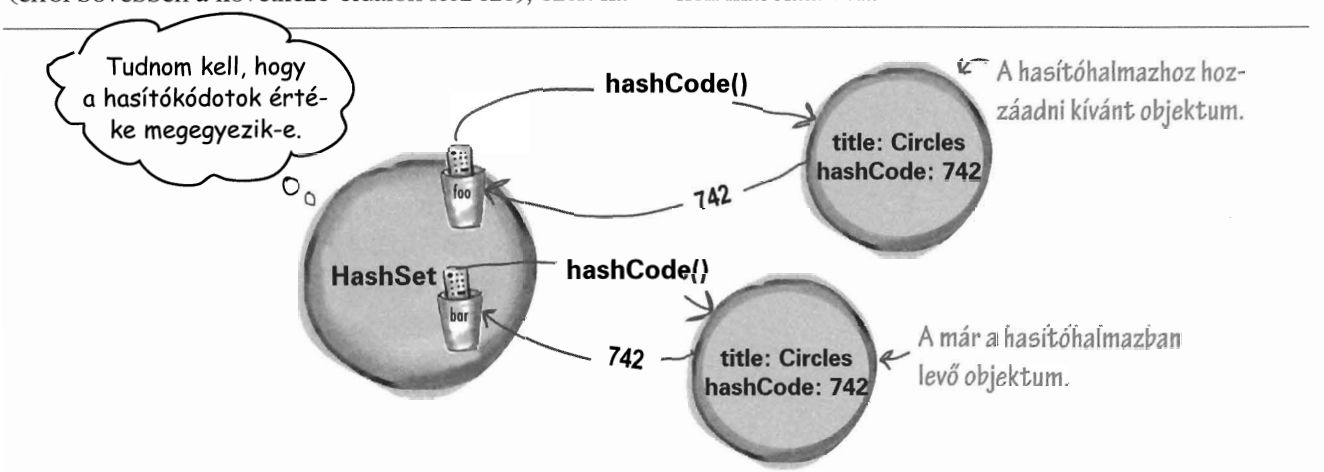
- Objektum**azonosság**
  Két hivatkozás, két objektum a kupacban – de az objektumok *azonos jelentésűnek* számítanak.
  Ha két különböző Song objektumot azonosként szeretnél kezelni (például mert úgy döntöttél, hogy két dalt azonosnak akarsz tekinteni, ha a *title* változójuk értéke megegyezik), felül kell bírálnod az Object osztálytól örökölt **hashCode()** és **equals()** tagfüggvényt *is*. Ahogy fentebb említettük, ha nem írod felül a hashCode()-ot, az (Object osztálytól örökölt) alapértelmezett viselkedés az, hogy minden objektum egyedi hasítókódot kap. Ahhoz tehát, hogy két egyenértékű objektum biztosan ugyanazt a hasítókódot adja vissza, muszáj felülbírálnod a hashCode() tagfüggvényt. Ezen kívül azonban az equals() tagfüggvényt is felül kell írnod, hogy mindig *true*-t adjon vissza, amikor meghívod az egyik objektumra (*bármelyikre* a kettő közül), és átadod neki a másik objektumot.

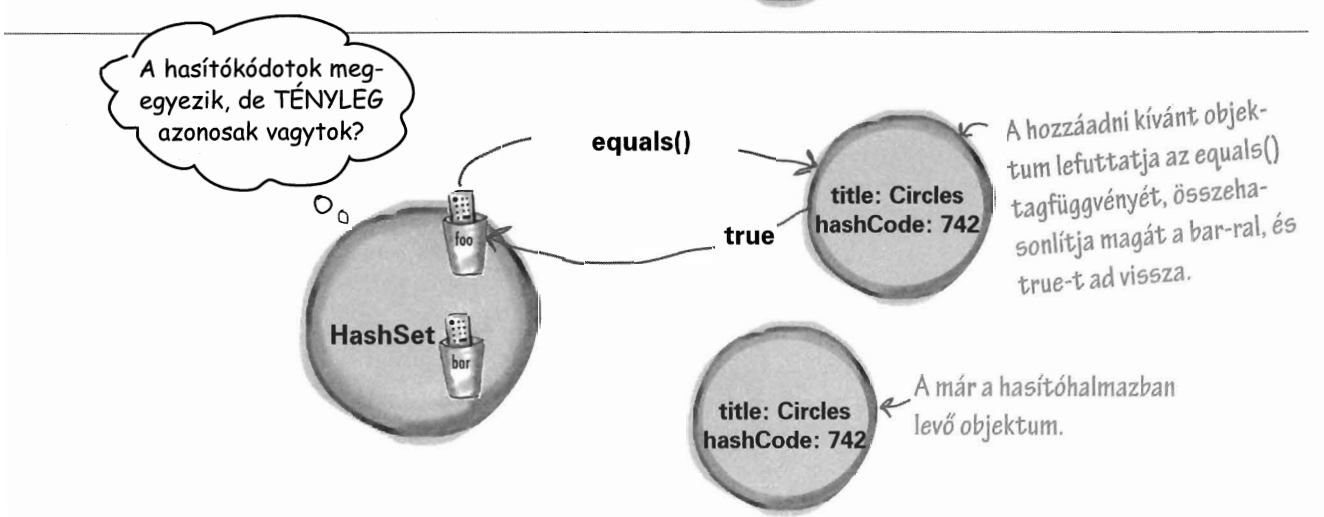



```
if (foo.equals(bar) && foo.hashCode() == bar.hashCode()) {
   // mindkét hivatkozás ugyanarra az objektumra
   // vagy két azonos objektumra mutat
}
```

A HashSet segédei az ismétlődések kiszűrésében: a hashCode() és az equals() tagfüggvény

Amikor egy objektumot helyezel egy hasítóhalmazba, a halmaz az objektum hasítókódjának értéke alapján határozza meg, hogy hová tegye az objektumot a halmazban. Emellett azonban az objektum hasítókódját a halmazban található összes többi objektum hasítókódjával is összeveti, és ha nem talál egyezőt, a HashSet feltételezi, hogy az új objektum nem másolat. Más szavakkal, ha a hasítókódok különböznek, a HashSet feltételezi, hogy az objektumok semmilyen módon nem lehetnek egyenlőek, ezért felül kell bírálnod a hashCode() tagfüggvényt, hogy az objektumok azonos értéket kaphassanak. Két azonos hasítókódú objektum azonban *nem* biztos, hogy egyenlő (erről bővebben a következő oldalon lesz szó), ezért ha a HashSet két objektumban – az újonnan beszúrt és egy már a halmazban levő objektumban – egyező hasítókódot talál, a halmaz meghívja az egyik objektum equals() tagfüggvényét, hogy lássa, *valóban* egyenlő-e a két azonos hasítókódú objektum. Ha egyenlőek, a HashSet tudja, hogy a hozzáadni próbált objektum a halmaz valamelyik elemének másolata, ezért a hozzáadást nem hajtja végre. Kivételt nem kapsz, de a HashSet add() tagfüggvénye visszaad egy logikai (boolean) értéket, ami jelzi (ha érdekel), hogy az új objektum hozzáadására sor került-e. Ha az add() tagfüggvényt *false* értéket ad vissza, tudhatod, hogy az új objektum a halmaz valamelyik meglevő elemének másolata volt.





A Song osztály a felülbírált hashCode() és equals() tagfüggvényekkel

```
class Song implements Comparable<Song>{
   String title;
   String artist;
   String rating;
   String bpm;

  public boolean equals(Object aSong) {
     Song s = (Song) aSong;
     return getTitle().equals(s.getTitle());
  }

  public int hashCode() {
      return title.hashCode();
  }

   public int compareTo(Song s) {
      return title.compareTo(s.getTitle());
   }

   Song(String t, String a, String r, String b) {
      title = t;
      artist = a;
      rating = r;
      bpm = b;
   }

   public String getTitle() {
       return title;
   }

   public String getArtist() {
      return artist;
   }

   public String getRating() {
      return rating;
   }

  public String getBpm() {
     return bpm;
  }

  public String toString() {
     return title;
  }
}
```

A HashSet (vagy bárki más, aki meghívta ezt a tagfüggvényt) átad egy újabb Song objektumot.

A JÓ hír az, hogy a cím egy String, a String-nek pedig van egy felülírt equals() tagfüggvénye. Így csak annyit kell tennünk, hogy megkérdezzük az egyik címtől, hogy azonos-e a másik dal címével.

Itt ugyanaz a helyzet... a String osztálynak van egy felülírt hashCode() tagfüggvénye, így csak vissza kell adni a címre meghívott hashCode()eredményét. Vedd észre, hogy a hashCode() és az equals() UGYANAZT a példányváltozót használja.

Most már működik! A hasítóhalmaz tartalmának kiírásakor nem kapunk ismétlődő elemeket. A sort()-ot azonban nem hívtuk meg újra, ezért amikor a tömblistát a hasítóhalmazba tettük, a HashSet nem őrizte meg a rendezési sorrendet.

File Edit Window Help RebootWindows

```
%java Jukebox6

[Pink Moon, Somersault, Shiva Moon, Circles,
Deep Channel, Passenger, Listen, Listen,
Listen, Circles]

[Circles, Circles, Deep Channel, Listen,
Listen, Listen, Passenger, Pink Moon, Shiva
Moon, Somersault]

[Pink Moon, Listen, Shiva Moon, Circles,
Deep Channel, Passenger, Somersault]
```

A Java objektumtörvénye a hashCode() és equals() tagfüggvényekre

Az Object osztály API-jának leírása a következő szabályok KÖTELEZŐ betartását írja elő:

- Ha két objektum azonos, akkor a hasítókódjuknak meg KELL egyeznie.
- Ha két objektum azonos, akkor az equals() a kettő közül bármelyikre meghívva true-t KELL, hogy visszaadjon. Más szavakkal, ha *a.equals(b)*, akkor *b.equals(a)*.
- Ha két objektum hasítókódjának azonos is az értéke, az objektumoknak NEM kötelező azonosnak lenniük. Ha viszont azonosak, a hasítókódjuk értékének azonosnak KELL lennie.
- Ha felülírod az equals() tagfüggvényt, akkor a hashCode()-ot is felül KELL írnod.
- A hashCode() alapértelmezett viselkedése az, hogy egyedi egész értékű azonosítószámot állít elő a kupacban levő minden objektum számára. Tehát ha nem bírálod felül a hashCode()-ot egy osztályban, akkor az adott típusba tartozó két objektum SOHA nem tekinthető azonosnak.
- Az equals() alapértelmezett viselkedése az, hogy == összehasonlítást végez. Más szavakkal, azt vizsgálja, hogy a két hivatkozás egyetlen objektumra mutat-e a kupacban. Tehát ha nem bírálod felül az equals() tagfüggvényt egy osztályban, akkor két objektum SOHA nem tekinthető azonosnak, mivel két különböző objektumra mutató hivatkozás mindig különböző bitmintát tartalmaz.

Az *a.equals(b)* egyben azt is kell, hogy jelentse, hogy
a.hashCode() == b.hashCode()
De az *a.hashCode() == b.hashCode()*
NEM kell, hogy *a.equals(b)*-t jelentsen

Nincsenek
hülye kérdések

K: Hogyan lehetséges, hogy a hasítókódok akkor is azonosak lehetnek, ha az objektumok nem azok?

V: A HashSet-ek a hasítókódokat arra használják, hogy az elemeket olyan módon tárolják, ami sokkal gyorsabbá teszi az elérésüket. Ha egy objektumot úgy próbálsz megtalálni egy ArrayList-ben, hogy a tömblistának az objektum másolatát adod át (és nem egy indexértéket), a tömblistának a lista elejétől kell kezdenie a keresést, és meg kell vizsgálnia minden elemet, hogy egyezést találjon. Egy HashSet ezzel szemben sokkal gyorsabban meg tud találni egy elemet, mert a hasítókódokkal mintegy felcímkézi az elemek tárolására használt „vödröket". Ha tehát azt mondod, hogy „Keresd meg azt az objektumot a halmazban, amelyik pontosan olyan, mint ez...", a hasítóhalmaz kiolvassa a hasítókód értékét a neki átadott Song-másolatból (legyen az érték mondjuk 742), és azt mondja: „Ó, pontosan tudom, hol tárolódik a 742-es hasítókódú objektum..." – és már nyúl is a 742-es számú vödörhöz.

Ez persze nem a teljes történet, amit egy számítástechnika órán hallanál, de ahhoz elég, hogy hatékonyan használd a hasítóhalmazokat. Egy jó hasítókód-algoritmus megalkotása valójában számos doktori disszertáció témája, így ebben a könyvben nem is akarunk jobban belemélyedni.

A lényeg az, hogy a hasítókódok anélkül is megegyezhetnek, hogy garantálnák az objektumok azonosságát, mert a hashCode() tagfüggvényben használt „hasítóalgoritmus" több objektumra is visszaadhatja ugyanazt az értéket. Igen, ez azt jelenti, hogy több objektum is ugyanabban a vödörben találhatja magát a hasítóhalmazban (mivel minden vödör egyetlen hasítókód-értéket jelképez), de ez még nem a világ vége. Csak annyit jelent, hogy a HashSet esetleg egy kicsit kevésbé lesz hatékony (vagy rendkívül nagy számú elemmel töltődik fel), de ha a HashSet egynél több objektumot talál ugyanazzal a hasítókóddal ellátott vödörben, egyszerűen megnézi az equals() tagfüggvénnyel, hogy tökéletes-e az egyezés. Más szavakkal, a hasítókód-értékek néha segítenek leszűkíteni a keresést, de a pontos találathoz a hasítóhalmaznak továbbra is fognia kell az adott (az azonos hasítókóddal rendelkező objektumok mindegyikét tároló) vödörben található összes objektumot, majd meg kell hívnia az equals() tagfüggvényt rájuk, hogy megállapítsa, hogy a keresett objektum az említett vödörben található-e.

Ha azt akarod, hogy a halmaz rendezett is maradjon, használj fahalmazt

A TreeSet (fahalmaz) annyiban hasonlít a HashSet-hez, hogy megakadályozza az ismétlődést – ugyanakkor a listát *rendezetten tartja.* A fahalmaz a sort() tagfüggvényhez hasonlóan működik: ha a halmaz argumentum nélküli konstruktorával hozol létre egy TreeSet-et, a TreeSet a rendezéshez az egyes objektumok compareTo() tagfüggvényét használja, de ha lehetőséged van átadni egy Comparator-t a TreeSet konstruktorának, akkor a fahalmaz azt fogja használni a compareTo() helyett. A fahalmazok hátránya, hogy ha *nincs szükséged* rendezésre, akkor ezért a szolgáltatásért némi sebességcsökkenéssel fizetsz. Ez a sebességcsökkenés ugyanakkor a legtöbb alkalmazásban szinte észrevehetetlen.

```
import java.util.*;
import java.io.*;
public class Jukebox8 {
   ArrayList<Song> songList = new ArrayList<Song>();
   int val;

   public static void main(String[] args) {
      new Jukebox8().go();
   }

   public void go() {
      getSongs();
      System.out.println(songList);
      Collections.sort(songList);
      System.out.println(songList);
      TreeSet<Song> songSet = new TreeSet<Song>();
      songSet.addAll(songList);
      System.out.println(songSet);
   }

   void getSongs() {
      try {
         File file = new File("SongListMore.txt");
         BufferedReader reader = new BufferedReader(new FileReader(file));
         String line =  null;
         while ((line= reader.readLine()) != null) {
              addSong(line);
         }

      } catch(Exception ex) {
         ex.printStackTrace();
      }
   }

   void addSong(String lineToParse) {
      String[] tokens = lineToParse.split("/");
      Song nextSong = new Song(tokens[0], tokens[1], tokens[2], tokens[3]);
      songList.add(nextSong);
   }
}
```

Egy HashSet helyett eegy TreeSet-példányt hozunk létre. Az argumentum nélküli TreeSet-konstruktor meghívása azt jelenti, hogy a halmaz a Song objektum compareTo() tagfüggvényét fogja használni a rendezéshez. (Persze egy Comparator-t is átadhattunk volna.)

Az addAll() segítségével az összes dalt a halmazhoz adhatjuk a HashSet-ből. (De azt is megtehetjük, hogy a dalokat egyenként adjuk hozzá a songSet.add() segítségével, ahogy az ArrayList esetében tettük.)

Amit tudnod KELL a fahalmazokról...

A TreeSet egyszerűnek tűnik, de fontos, hogy valóban értsd, hogy mit kell tenned ahhoz, hogy használhasd. Ezt annyira lényegesnek éreztük, hogy egy gyakorlatot is összeállítottunk, hogy *muszáj* legyen elgondolkodnod rajta. NE lapozz addig, amíg ezt a feladatot el nem végezted. *Komolyan.*



Hegyezd ki a ceruzád!

Nézd meg ezt a kódot. Olvasd el figyelmesen, aztán válaszolj az alatta található kérdésekre. (Megjegyzés: a kódban nincsenek nyelvtani hibák.)

```
import java.util.*;

public class TestTree {
   public static void main (String[] args) {
      new TestTree().go();
   }

   public void go() {
     Book b1 = new Book("How Cats Work");
     Book b2 = new Book("Remix your Body");
     Book b3 = new Book("Finding Emo");

     TreeSet<Book> tree = new TreeSet<Book>();
     tree.add(b1);
     tree.add(b2);
     tree.add(b3);
     System.out.println(tree);
   }
}

class Book {
   String title;
   public Book(String t) {
     title = t;
   }
}
```

1) Milyen eredményt kapsz, ha lefordítod ezt a kódot?

2) Ha a kódot sikerült lefordítani, milyen eredményt kapsz a TestTree osztály futtatásakor?

3) Ha probléma lép fel (akár fordításkor, akár futáskor) a fenti kóddal, hogyan javítanád ki?

A fahalmazok elemeinek összehasonlíthatónak KELL lenniük

Egy fahalmaz nem képes kiolvasni a programozó agyából, hogy az objektumokat hogyan kell rendezni – meg kell mondanod neki.

Egy TreeSet használatához az alábbiak egyikének igaznak kell lennie:

- **A lista elemeinek olyan típusba kell tartozniuk, amely megvalósítja a *Comparable* felületet**

  Az előző oldalon szereplő Book osztály nem valósította meg a Comparable felületet, ezért futásidőben nem működik. Gondold végig: szegény fahalmaznak az az egyetlen életcélja, hogy rendezetten tartsa az elemeket, de még egyszer: fogalma sincs, hogyan rendezze a Book objektumokat! Fordításkor nem okoz hibát, mert a TreeSet add() tagfüggvénye nem összehasonlítható (Comparable) típust vár – bármilyen típust elfogad, amit a fahalmaz létrehozásakor használtál. Más szavakkal, ha azt mondod, hogy new TreeSet<Book>(), akkor az add() tagfüggvény jelentése lényegében add(Book). Nincs olyan követelmény, hogy a Book osztálynak meg kell valósítania a Comparable felületet! Futásidőben azonban kudarcot vall, amikor a második elemet megpróbálod hozzáadni a halmazhoz. Ez az első alkalom, hogy a halmaz megkísérli meghívni az egyik objektum compareTo() tagfüggvényét – és nem tudja.

```
class Book implements Comparable {
   String title;
   public Book(String t) {
     title = t;
   }
   public int compareTo(Object b) {
     Book book = (Book) b;
     return (title.compareTo(book.title));
   }
}
```

VAGY

- **A TreeSet-nek azt a túlterhelt konstruktorát használod, amelyik egy *Comparator*-t vár**

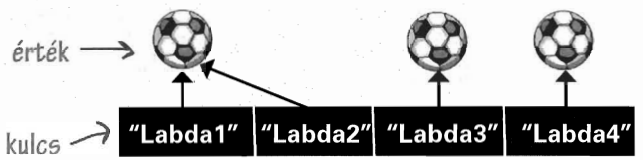
  A TreeSet nagyon hasonlóan működik, mint a sort() tagfüggvény: választhatsz, hogy az elem compareTo() tagfüggvényét használod, feltételezve, hogy az elem típusa megvalósítja a Comparable felületet, VAGY használhatsz egyedi Comparator-t, amely tudja, hogyan kell rendezni a halmaz elemeit. Ha egyedi összehasonlítót szeretnél használni, a TreeSet Comparator-t váró konstruktorát kell meghívnod.

```
public class BookCompare implements Comparator<Book> {
   public int compare(Book one, Book two) {
      return (one.title.compareTo(two.title));
   }
}

class Test {
     public void go() {
     Book b1 = new Book("How Cats Work");
     Book b2 = new Book("Remix your Body");
     Book b3 = new Book("Finding Emo");
     BookCompare bCompare = new BookCompare();
     TreeSet<Book> tree = new TreeSet<Book>(bCompare);
     tree.add(new Book("How Cats Work");
     tree.add(new Book("Finding Emo");
     tree.add(new Book("Remix your Body");
     System.out.println(tree);
   }
}
```

Listát és halmazt már láttunk, jöjjön egy térkép!

A listák (List) és a halmazok (Set) nagyszerűek, de néha egy térkép (Map) bizonyul a legjobb gyűjteménynek. (Ne feledd: a térképek a Java-gyűjtemények közé tartoznak, de nem valósítják meg a Collection – „Gyűjtemény" – felületet.) Tegyük fel, hogy egy olyan gyűjteményt szeretnél, amely úgy viselkedik, mint egy tulajdonságlista: átadsz neki egy nevet, és ő visszaadja az adott névhez társított értéket. Bár a kulcsok gyakran karakterláncok (String), bármilyen Java-objektum (automatikus dobozoláson keresztül pedig akár egy primitív is) használható kulcsként.



Map

Egy Map minden eleme valójában KÉT objektum: egy *kulcs* és egy *érték*. Ismétlődő *értékek* lehetnek benne, de ismétlődő kulcsok NEM.

Map – példa

```
import java.util.*;

public class TestMap {

   public static void main(String[] args) {

     HashMap<String, Integer> scores = new HashMap<String, Integer>();

     scores.put("Kathy", 42);
     scores.put("Bert", 343);
     scores.put("Skyler", 420);

      System.out.println(scores);
      System.out.println(scores.get("Bert"));
   }
}
```

A HashMap KÉT típusparamétert igényel – egyet a kulcs, és egyet az érték számára.

Az add() helyett a put()-ot használjuk, természetesen most már két argumentummal (kulcs, érték).

A get() tagfüggvény egy kulcsot vár, és az értéket adja vissza (ami ebben az esetben egy egész szám).

```
File Edit Window Help WhereAmI
%java TestMap

{Skyler=420, Bert=343, Kathy=42}
343
```

Amikor kiíratsz egy térképet, a „kulcs=érték" párok kapcsos zárójelek ({}) között jelennek meg a listák és halmazok kiírásánál látott szögletes zárójelek ([]) helyett.

Vissza az általánosításhoz

Emlékezz vissza, hogy a fejezetben korábban azt mondtuk, hogy az általános típusú argumentumokat fogadó tagfüggvények *furcsák* lehetnek. A „furcsa" szót itt többalakú értelemben használjuk. Ha ez nem lenne tiszta, ne törődj vele, csak olvass tovább – a kifejtés igényel még pár oldalt.
Egy emlékeztetővel kezdjük a *tömb*argumentumok többalakú működéséről, majd megnézzük, hogyan fest ugyanez általános listákkal.
Az alábbi kód hiba nélkül lefordítható és futtatható:

Ha egy tagfüggvény-argumentum Animal objektumok *tömbje*, akkor bármilyen Animal-altípusba tartozó tömböt elfogad.
Más szavakkal, ha egy tagfüggvény bevezetése így fest...

void foo(Animal[] a) {}

...akkor az alábbi két hívás egyaránt megengedett (feltételezve, hogy a Dog az Animal bővítése):

foo(anAnimalArray);
foo(aDogArray);

Így működik szokványos tömbökkel:

```
import java.util.*;

public class TestGenerics1 {
   public static void main(String[] args) {
      new TestGenerics1().go();
   }

   public void go() {
      Animal[] animals = {new Dog(), new Cat(), new Dog()};
      Dog[] dogs = {new Dog(), new Dog(), new Dog()};
      takeAnimals(animals);
      takeAnimals(dogs);
   }

  public void takeAnimals(Animal[] animals) {
      for(Animal a: animals) {
        a.eat();
      }
   }
}
```

Bevezetünk és létrehozunk egy Animal tömböt, ami egyaránt tárol kutyákat (Dog) és macskákat (Cat).

Bevezetünk és létrehozunk egy Dog tömböt, ami csak kutyákat tárol (a fordító nem engedi, hogy macskákat tegyünk bele).

A lényeg az, hogy a takeAnimals() tagfüggvény Animal[] és Dog[] tömböt is elfogad, mivel a Dog egyfajta Animal. Így működik a többalakúság.

Ne feledd, hogy CSAK az Animal típusban bevezetett tagfüggvényeket hívhatjuk meg, mivel az animals paraméter Animal tömb típusú, és nem végezünk típusátalakítást. (Mire is alakítanánk? Ez a tömb kutyákat és macskákat is tud tárolni.)

```
abstract class Animal {
   void eat() {
     System.out.println("animal eating");
   }
}
class Dog extends Animal {
   void bark() { }
}
class Cat extends Animal {
   void meow() { }
}
```

Az Animal egyszerűsített osztályhierarchiája.

Többalakú argumentumok és általánosítások használata

Láttuk tehát, hogyan működik a dolog tömbökkel, de akkor is ugyanígy fog működni, ha tömbről tömblistára (ArrayList) váltunk? Ésszerű lenne, nem? Először próbáljuk csak az Animal-t tömblistává tenni. Csak pár dolgot kell változtatnunk a go() tagfüggvényen:

Csak ArrayList<Animal>-t átadva:

Egyszerű módosítás Animal[]-ról ArrayList<Animal>-re.

```
public void go() {
    ArrayList<Animal> animals = new ArrayList<Animal>();
    animals.add(new Dog());
    animals.add(new Cat());
    animals.add(new Dog());

    takeAnimals(animals);
}
```

Egyenként kell hozzáadnunk őket, mert itt nincs olyan rövid írásmód, mint a tömbök létrehozásánál.

Ez ugyanaz a kód, csak éppen az „animals" változó most egy tömb helyett egy tömblistára hivatkozik.

```
public void takeAnimals(ArrayList<Animal> animals) {
    for(Animal a: animals) {
    a.eat();
   }
}
```

A tagfüggvény most egy tömb helyett egy tömblistát vár, de minden más megegyezik. Ne feledd, hogy a for ciklusok tömbökkel és gyűjteményekkel is használhatók.

Sikeresen lefordítható és futtatható

```
File Edit Window Help CatFoodIsBetter
%java TestGenerics2

animal eating
animal eating
animal eating
animal eating
animal eating
animal eating
```

ArrayList<Dog> esetén is működik?

A többalakúság miatt a fordító megengedi, hogy egy Dog tömböt adjunk át egy olyn tagfüggvénynek, amely Animal tömbargumentummal rendelkezik – semmi gond. Egy ArrayList<Animal> is átadható egy ArrayList<Animal> argumentumú tagfüggvénynek. A nagy kérdés tehát az, hogy az ArrayList<Animal> argumentum elfogad-e egy ArrayList<Dog>-ot? Ha tömbökkel működik, itt is kell neki, nem?

Csak ArrayList<Dog>-ot átadva:

```
public void go() {
   ArrayList<Animal> animals = new ArrayList<Animal>();
   animals.add(new Dog());
   animals.add(new Cat());
   animals.add(new Dog());
   takeAnimals(animals);
   ArrayList<Dog> dogs = new ArrayList<Dog>();
   dogs.add(new Dog());
   dogs.add(new Dog());
   takeAnimals(dogs);
}

public void takeAnimals(ArrayList<Animal> animals) {
     for(Animal a: animals) {
     a.eat();
    }
 }
```

Tudjuk, hogy ez a sor rendben működött.

Létrehozunk egy Dog tömblistát, és beletesszünk néhány kutyát.

Működni fog ez így is, hogy tömbről tömblistára váltottunk?

Fordításkor ezt kapjuk:

```
File Edit Window Help CatsAreSmarter
%java TestGenerics3

TestGenerics3.java:21: takeAnimals(java.util.
ArrayList<Animal>) in TestGenerics3 cannot be applied to
(java.util.ArrayList<Dog>)
     takeAnimals(dogs);
     ^
1 error
```

Nagyon félresiklott, pedig olyan jónak tűnt...



És ezt nekem el kéne fogadnom? Teljesen tönkreteszi az állatszimulációs programomat, amelynek bármilyen állatot el kell fogadnia, hogy a kutya- és macskatenyésztők kutya- és macskalistákat küldhessenek az állatorvosnak. Most azt mondod, hogy ez nem lehetséges, ha tömbök helyett gyűjteményeket használok?

Mi lenne, ha lehetséges lenne...?

Tegyük fel, hogy a fordító nem gördít akadályokat eléd, és megengedi, hogy átadj egy ArrayList<Dog>-ot egy tagfüggvénynek, amit így vezettél be:

```
public void takeAnimals(ArrayList<Animal> animals) {
   for(Animal a: animals) {
      a.eat();
   }
}
```

Semmi nincs ebben a tagfüggvényben, ami ártalmasnak *tűnik*, nem igaz? Végül is, a többalakúságnak éppen az a lényege, hogy egy Dog is megcsinálhat bármit, amire egy Animal képes (ebben az esetben futtathatja az eat() tagfüggvényt). De akkor mi a gond azzal, ha az eat() tagfüggvényt minden Dog hivatkozásra meghívjuk?

Semmi. Nincs semmilyen probléma.

A *fenti* kóddal nincs semmi baj. De az *alábbi* kóddal már más a helyzet:

```
public void takeAnimals(ArrayList<Animal> animals) {
   animals.add(new Cat());
}
```

← Aú! Most szúrtunk be egy macskát egy csak kutyáknak fenntartott tömblistába...

Na, ez a gond. Egy Cat-et természetesen nyugodtan hozzáadhatsz egy ArrayList<Animal>-hoz. Egy Amimal-hoz hasonló alaptípusba tartozó tömblistának éppen ez a célja: hogy bármilyen állatfajtát beletehess egyetlen Animal típusú ArrayList-be.
Ezzel szemben ha az Animal típusú ArrayList-et váró tagfüggvénynek egy Dog típusú – KIZÁRÓLAG Dog-ok tárolására tervezett – ArrayList-et adsz át, akkor előfordulhat, hogy hirtelen egy macska kerül a kutyalistába. A fordító tudja, hogy ha megengedné neked, hogy egy ilyen tagfüggvénynek egy Dog ArrayList-et adj át, akkor megtörténhetne, hogy valaki futásidőben egy macskát tesz a kutyalistába. Ezért a fordítóprogram nem is hagyja, hogy ilyen kockázatot vállalj.

Ha egy tagfüggvényt ArrayList<Animal>argumentummal vezetsz be, akkor KIZÁRÓLAG ArrayList<Animal> adható át neki, ArrayList<Dog> vagy ArrayList<Cat> nem.



Várjunk csak egy percet... Ha ezért nem lehet egy Dog ArrayList-et átadni egy Animal ArrayList-et váró tagfüggvénynek – hogy ne lehessen macskát tenni egy kutyalistába –, akkor a tömbökkel miért működik? A tömbökkel nem ugyanez a probléma? Egy Dog[] tömbhöz nem lehet Cat objektumot adni?

A tömbök típusát a Java futásidőben is ellenőrzi, a gyűjteményekét azonban csak fordításkor

Tegyük fel, hogy tényleg hozzáadsz egy Cat-et egy Dog[]-ként bevezetett tömbhöz (egy olyan tömbhöz, amelyet egy Animal[]-ként bevezetett tagfüggvény-argumentumnak adtál át, ami a tömbök esetében teljesen szabályos):

```
public void go() {
   Dog[] dogs = {new Dog(), new Dog(), new Dog()};
   takeAnimals(dogs);
}

public void takeAnimals(Animal[] animals) {
   animals[0] = new Cat();
}
```

Itt egy macskát teszünk egy kutyatömbbe. A fordító megengedi, mert tudja, hogy a tagfüggvénynek Cat és Animal tömböt is átadhattál, ezért lehetséges, hogy ez rendben van.

Ez lefordítható, de amikor futtatod, ezt kapod:

Húha! De legalább a JVM leállította...

```
File Edit Window Help CatsAreSmarter
%java TestGenerics1
Exception in thread "main" java.lang.ArrayStoreException:
Cat
        at TestGenerics1.takeAnimals(TestGenerics1.java:16)
        at TestGenerics1.go(TestGenerics1.java:12)
        at TestGenerics1.main(TestGenerics1.java:5)
```


Hát nem lenne csodás, ha mégis volna rá mód, hogy többalakú gyűjteménytípusokat használjunk tagfüggvény-argumentumként, hogy az állatorvosi program kutya- és macska-listákat is elfogadhasson? Így egy ciklussal végig-mehetnénk a listákon, és meghívhatnánk az immunize() tagfüggvényüket, miközben azért továbbra is ügyelnénk arra, hogy ne lehessen macskát tenni a kutyalistába. Meseszép lenne. De ez csak álom...

Helyettesítő karakterek – a felmentő sereg

Szokatlan formájú ugyan, de *van* rá mód, hogy olyan tagfüggvény-argumentumot adj meg, amely bármilyen Animal-altípusú ArrayList-et elfogad. A legegyszerűbb, ha egy **helyettesítő karaktert** (wildcard) használsz, amelyet kifejezetten ezért adtak a Java nyelvhez:

```
public void takeAnimals(ArrayList<? extends Animal> animals) {
    for(Animal a: animals) {
        a.eat();
    }
}
```

Ne feledd, hogy az „extends" jelentése itt extends VAGY implements is lehet, a típustól függően. Tehát ha mondjuk olyan típusú tömblistákat szeretnél elfogadni, amelyek a Pet felületet valósítják meg, a bevezetést így kell megírnod: ArrayList<? extends Pet>

Most nyilván felmerül benned a kérdés: „Mi a *különbség*? Nem ugyanaz a probléma, mint korábban? A fenti tagfüggvény nem csinál semmi veszélyeset – egy olyan tagfüggvényt hívunk meg, amellyel minden Animal-altípus garantáltan rendelkezik –, de továbbra is megeshet, hogy valaki úgy módosítja, hogy egy Cat-et adjon hozzá az *animals* listához, pedig valójában egy ArrayList<Dog>-ról van szó, nem igaz? És mivel a Java futásidőben nem ellenőrzi ismét, miért lenne ez más, mint a helyettesítő karakter nélküli bevezetés?"
A kérdés jogos. A válasz azonban az, hogy a helyzet NEM ugyanaz. Amikor a deklarációban a <?> helyettesítő karaktert használod, a fordító egyáltalán nem enged hozzáadni a listához!

Ha egy tagfüggvény-argumentumban helyettesítő karaktert használsz, a fordító MEGAKADÁLYOZZA, hogy bármi olyasmit csinálj, ami megsértheti a tagfüggvény-paraméter által hivatkozott listát.
A lista elemeire továbbra is meghívhatsz tagfüggvényeket, de elemeket nem adhatsz a listához.
Más szavakkal, a lista elemeivel végezhetsz műveleteket, *új* dolgokat azonban nem vehetsz fel a listába.
Így futásidőben biztonságban vagy, mert a fordítóprogram semmi olyat nem enged csinálni, ami futáskor hibát okozna.
Vagyis ez MEGENGEDETT a takeAnimals() belsejében:

```
for(Animal a: animals) {
    a.eat();
}
```

EZ viszont nem fordítható le:

```
animals.add(new Cat());
```

Ugyanaz, másik utasításformával

Valószínűleg emlékszel rá, hogy amikor megnéztük a sort() tagfüggvényt, az egy általános típust használt, de szokatlan formában: a típusparaméter bevezetése a visszatérési típusé előtt szerepelt. Ez csupán egy másik módja a típusparaméter bevezetésének, az eredménye azonban ugyanaz:

Ez...

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

...ugyanazt csinálja, mint ez:

```
public void takeThing(ArrayList<? extends Animal> list)
```

Nincsenek hülye kérdések

K: Ha mindkettő ugyanazt csinálja, akkor mi dönti el, hogy melyiket kell választani?

V: Minden attól függ, hogy a „T" akarod-e máshol is használni. Mi a helyzet például akkor, ha azt szeretnéd, hogy a tagfüggvénynek két argumentuma legyen, méghozzá mindkettő az Animal-t bővítő típusú lista? Ebben az esetben hatékonyabb egyszer bevezetni a típusparamétert:

```
public <T extends Animal> void takeThing(ArrayList<T> one, ArrayList<T> two)
```

Ehelyett:

```
public void takeThing(ArrayList<? extends Animal> one,
                      ArrayList<? extends Animal> two)
```



Gyakorlat

Légy a fordító! – Haladóknak

A feladatod az, hogy eljátszd a fordítóprogram szerepét, és kiderítsd, hogy az alábbi utasítások közül melyeket lehet sikeresen lefordítani. Szerepelnek közöttük olyanok is, amelyekkel nem foglalkoztunk a fejezetben, ezért a válaszokat a TANULTAK alapján neked kell megtalálnod, a „szabályokat" új helyzetekre alkalmazva. Egyes esetekben találgatásra kényszerülhetsz, de ez nem baj – a lényeg az, hogy az eddigi ismereteidre támaszkodva elfogadható válasszal állj elő.



(Megjegyzés: feltételezzük, hogy a kódok érvényes osztályon és tagfüggvényen belül találhatók.)

Lefordítható?

☐
```
ArrayList<Dog> dogs1 = new ArrayList<Animal>();
```

☐
```
ArrayList<Animal> animals1 = new ArrayList<Dog>();
```

☐
```
List<Animal> list = new ArrayList<Animal>();
```

☐
```
ArrayList<Dog> dogs = new ArrayList<Dog>();
```

☐
```
ArrayList<Animal> animals = dogs;
```

☐
```
List<Dog> dogList = dogs;
```

☐
```
ArrayList<Object> objects = new ArrayList<Object>();
```

☐
```
List<Object> objList = objects;
```

☐
```
ArrayList<Object> objs = new ArrayList<Dog>();
```

A „Fordított programozás” ceruzahegyező gyakorlat megoldása

```
import java.util.*;

public class SortMountains {

  LinkedList<Mountain> mtn = new LinkedList<Mountain>();

  class NameCompare implements Comparator <Mountain> {
    public int compare(Mountain one, Mountain two) {
      return one.name.compareTo(two.name);
    }
  }
  class HeightCompare implements Comparator <Mountain> {
    public int compare(Mountain one, Mountain two) {
      return (two.height - one.height);
    }
  }
  public static void main(String [] args) {
    new SortMountain().go();
  }
  public void go() {
    mtn.add(new Mountain("Longs", 14255));
    mtn.add(new Mountain("Elbert", 14433));
    mtn.add(new Mountain("Maroon", 14156));
    mtn.add(new Mountain("Castle", 14265));
    System.out.println("as entered:\n" + mtn);
    NameCompare nc = new NameCompare();
    Collections.sort(mtn, nc);
    System.out.println("by name:\n" + mtn);
    HeightCompare hc = new HeightCompare();
    Collections.sort(mtn, hc);
    System.out.println("by height:\n" + mtn);
  }
}

class Mountain {
  String name;
  int height;

  Mountain(String n, int h) {
    name = n;
    height = h;
  }
  public String toString( ) {
    return name + " " + height;
  }
```

Észrevetted, hogy a magasságok (height) listája CSÖKKENŐ sorrendű? :)

Kimenet:

```
File  Edit  Window  Help  ThisOne'sForBob
%java SortMountains
as entered:
[Longs 14255, Elbert 14433, Maroon 14156, Castle 14265]
by name:
[Castle 14265, Elbert 14433, Longs 14255, Maroon 14156]
by height:
[Elbert 14433, Castle 14265, Longs 14255, Maroon 14156]
```

Gyakorlat – Megoldás

Lehetséges válaszok:

Comparator
Comparable
compareTo()
compare()
igen
nem

Ha adott a következő lefordítható utasítás:

```
Collections.sort(myArrayList);
```

1. Mit kell a `myArrayList`-ben tárolt objektumok osztályának megvalósítania? **Comparable**
2. Melyik tagfüggvényt kell a `myArrayList`-ben tárolt objektumok osztályának megvalósítania? **compareTo()**
3. A `myArrayList`-ben tárolt objektumok osztálya megvalósíthatja a `Comparator` ÉS a `Comparable` felületet is? **igen**

Ha adott a következő lefordítható utasítás:

```
Collections.sort(myArrayList, myCompare);
```

4. A `myArrayList`-ben tárolt objektumok osztálya megvalósíthatja a `Comparable` felületet? **igen**
5. A `myArrayList`-ben tárolt objektumok osztálya megvalósíthatja a `Comparator`-t? **igen**
6. A `myArrayList`-ben tárolt objektumok osztályának meg kell valósítania a `Comparable` felületet? **nem**
7. A `myArrayList`-ben tárolt objektumok osztályának meg kell valósítania a `Comparator`-t? **nem**
8. Mit kell megvalósítania a `myCompare` objektum osztályának? **Comparator**
9. Melyik tagfüggvényt kell megvalósítania a `myCompare` objektum osztályának? **compare()**

Légy a fordító! – Megoldás



Lefordítható?

- [ ] `ArrayList<Dog> dogs1 = new ArrayList<Animal>();`
- [ ] `ArrayList<Animal> animals1 = new ArrayList<Dog>();`
- [x] `List<Animal> list = new ArrayList<Animal>();`
- [ ] `ArrayList<Dog> dogs = new ArrayList<Dog>();`
- [ ] `ArrayList<Animal> animals = dogs;`
- [x] `List<Dog> dogList = dogs;`
- [x] `ArrayList<Object> objects = new ArrayList<Object>();`
- [x] `List<Object> objList = objects;`
- [ ] `ArrayList<Object> objs = new ArrayList<Dog>();`

A kód közzététele



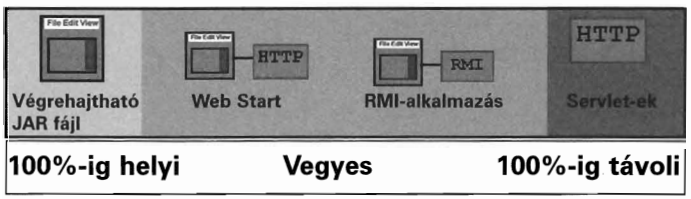
Ideje elengedned a kódodat. Megírtad, tesztelted, finomítottál rajta, minden ismerősöddel tudattad, hogy nem bánnád, ha soha többé nem látnál egyetlen kódsort sem – de végül is egy műalkotást hoztál létre. A program fut! De hogyan tovább? *Hogyan* jutattod el a végfelhasználókhoz? És pontosan *mit* is adsz oda nekik? Mi van, ha azt sem tudod, kik a végfelhasználók? Könyvünk két utolsó fejezetében azt járjuk körül, hogy miként szervezheted/rendszerezheted, csomagolhatod és teheted közzé a Java-kódjaidat. A helyi, részben helyi és távoli közzétételi lehetőségeket egyaránt górcső alá vesszük, beleértve a végrehajtható JAR („befőttesüveg") fájlokat, a Java Web Start-ot, az RMI-t és a kiszolgálói kisalkalmazásokat (servlet-eket) is. Ebben a fejezetben az idő legnagyobb részét a kód rendszerezésével és csomagolásával fogod tölteni – azokat a dolgokat tanulod meg, amelyekre a végleges közzétételi módtól függetlenül szükséged lesz –, míg az utolsó fejezetben az egyik legklasszabb dologgal zárod a tanulmányaidat, amire csak a Java lehetőséget nyújt: lazítással. A közzététellel még nem búcsúzol el örökre a kódtól – karbantartásra mindig szükség van…

Az alkalmazás közzététele

Pontosan *mi* is egy Java-alkalmazás? Másként fogalmazva: ha készen vagy a fejlesztéssel, mit kell közzétenned (deploy)? Valószínű, hogy a végfelhasználóid rendszere nem egyezik meg pontosan a tiéddel – és ami még fontosabb, nekik nincs birtokukban az alkalmazásod. Ideje hát, hogy a programodat közzétételre alkalmas formába öntsd a Külvilág számára. Ebben a fejezetben a helyi közzétételt vizsgáljuk, beleértve a végrehajtható (executable) JAR fájlok használatát, valamint a Java Web Start nevű, részben helyi, részben távoli közzétételi módszert. A következő fejezetben aztán a valódi távoli közzétételi lehetőségekkel ismerkedhetsz meg, köztük az RMI-vel és a kiszolgálói kisalkalmazásokkal (servlet-ekkel).

A Java-programok különféle osztályokból állnak: ezek jelentik a fejlesztés kimenetét.
Az igazi kérdés az, hogy mit kezdj ezekkel az osztályokkal, amikor végeztél.

Közzétételi lehetőségek



1. **Helyi**
   A teljes alkalmazás a végfelhasználó számítógépén fut, önálló (valószínűleg grafikus felhasználói felületű) programként, végrehajtható JAR fájl formájában telepítve (a JAR fájlokkal pár oldallal később foglalkozunk részletesen).

2. **Vegyesen helyi és távoli**
   Az alkalmazás ügyfélrésze a felhasználó helyi rendszerén fut, egy kiszolgálóhoz kapcsolódva, ahol az alkalmazás többi része működik.

3. **Távoli**
   A teljes Java-alkalmazás egy kiszolgálórendszeren fut, és az ügyfél valamilyen nem Java eszközzel (valószínűleg egy böngészőn keresztül) fér hozzá.

Mielőtt azonban igazán elmélyednénk a közzétételben, lépjünk vissza egy kicsit, és nézzük meg, mi történik, amikor befejezted az alkalmazásod programozását, és egyszerűen fogni szeretnéd az osztályfájlokat, és odaadni egy végfelhasználónak.
Mi van valójában a *munkakönyvtárban*?



Agygyúró

Mik az előnyei és mik a hátrányai annak, ha egy Java-programot a végfelhasználó számítógépén futó helyi, önálló alkalmazásként működtetsz?
Mik az előnyei és mik a hátrányai annak, ha a Java-programodat web alapú rendszerként üzemelteted, amelyben a felhasználó egy webböngészővel tartja a kapcsolatot, és a Java-kód kiszolgálói kisalkalmazások formájában fut a kiszolgálón?



Képzeld el ezt a forgatókönyvet...

Bob vígan dolgozik klassz új Java-programja utolsó darabjain. Miután heteket töltött „csak még egy fordítás" módban, most már tényleg karnyújtásnyira van a végétől. A program egy meglehetősen kifinomult grafikus felületű alkalmazás, de mivel az oroszlánrésze Swing-kód, csak kilenc saját osztályt tartalmaz.
Végül eljön az idő, amikor a programot le kell szállítani az ügyfélnek. Bob úgy véli, csak annyit kell tennie, hogy lemásolja a kilenc osztályfájlt, hiszen az ügyfél már telepítette a Java API-t. Így aztán azzal kezdi, hogy kiad egy `ls` („listázás") parancsot a fájljait tároló könyvtárra...



Húha! Itt valami furcsa dolog történt. 18 állomány (9 forráskódfájl és 9 lefordított osztályfájl) helyett Bob 31 fájlt lát, amelyek közül soknak nagyon furcsa a neve:

Account$FileListener.class
Chart$SaveListener.class

...és így tovább. Bob teljesen megfeledkezett róla, hogy a fordítónak minden általa készített GUI-eseményfigyelő belső osztály számára is elő kell állítania osztályfájlokat – ezek lennének a furcsa nevű osztályok.
Bobnak tehát gondosan ki kell válogatnia azokat az osztályfájlokat, amelyekre szüksége van. Ha csak egyet is kihagy, a programja nem fog működni. A dolog azonban nem egyszerű, mert nem szeretné, ha véletlenül egy *forráskódfájlt* küldene el az ügyfélnek, csakhogy minden egyetlen könyvtárban egymásra hányva található.

A forráskód- és osztályfájlok szétválasztása

Ha egyetlen könyvtár tárol egy nagy kupac forráskód- és osztályfájlt, akkor nehéz eligazodni a rendetlenségben. Bob jobban járt volna, ha már kezdettől fogva rendszerezi a fájljait, és a forráskódokat, illetve a lefordított kódokat külön tárolja. Más szavakkal, gondoskodnia kellett volna róla, hogy a lefordított osztályfájljai ne ugyanabba a könyvtárba kerüljenek, mint a forráskódjai.

A kulcsot a könyvtárszerkezet és a `-d` fordítói kapcsoló kombinálása jelenti.

A fájljaidat több tucatnyi módon rendszerezheted, és a cégednek is meglehet a saját módszere, amit rád erőltet. Mi ugyanakkor egy olyan rendszerezési sémát javaslunk, ami szinte szabványosnak számít.

Ebben a sémában először egy projektkönyvtárat hozol létre, majd ezen belül egy **source** (forrás) és egy **classes** (osztályok) nevű alkönyvtárat. A forráskódot (a .java fájlokat) a **source** könyvtárba mented, majd a kódokat úgy fordítod le, hogy a kimenet (a .class fájlok) a ***classes*** könyvtárba kerüljön.

Ezt egy hasznos fordítói kapcsoló, a **`-d`** segíti.



Fordítás a –d (directory, vagyis könyvtár) kapcsolóval

```
%cd MyProject/source
%javac  -d ../classes  MyApp.java
```

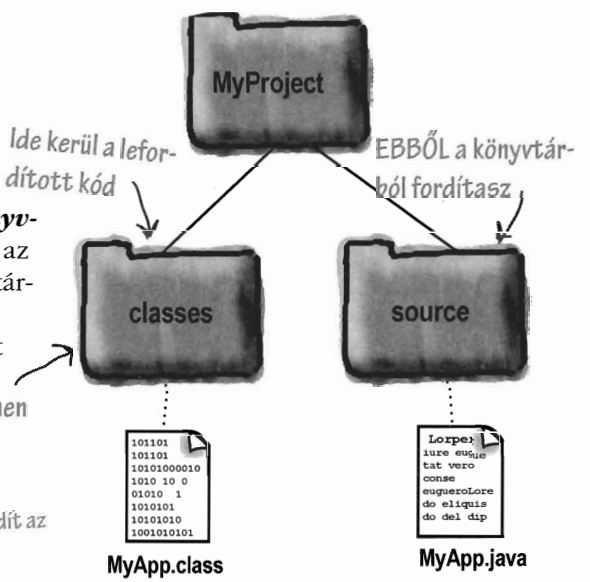
Ez arra utasítja a fordítót, hogy a lefordított kódot (az osztályfájlokat) a „classes" könyvtárba tegye, amely a jelenlegi munkakönyvtár alatt található.

Az utolsó dolog továbbra is a lefordítandó java fájl neve.

A **`-d`** kapcsoló használatával *te* döntheted el, hogy melyik ***könyvtárban*** kössön ki a lefordított kód, tehát nem kell elfogadnod az alapértelmezést, miszerint az osztályfájlok ugyanabba a könyvtárba kerülnek, mint ahol a forráskód is található. Ha a source könyvtárban található összes .java fájlt le akarod fordíttatni, ezt kell írnod:

```
%javac  -d ../classes  *.java
```

A * .java utasítás MINDEN forrásfájlt lefordít az aktuális könyvtárban.



A kód futtatása

```
%cd MyProject/classes
%java Mini
```

A program futtatása a „classes" könyvtárból

(Megjegyzés a hibaelhárításhoz: ebben a fejezetben végig azt feltételezzük, hogy az aktuális munkakönyvtár – vagyis a „." – szerepel az osztályelérési útvonalon (classpath). Ha kifejezetten beállítod a classpath környezeti változót, ügyelj rá, hogy szerepeljen benne a „.".)

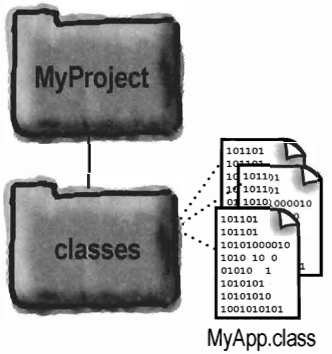
Tedd a Java-kódot egy befőttesüvegbe!



A **JAR** („befőttesüveg") a **J**ava **AR**chive (Java-archívum) rövidítése. A JAR a pkzip fájlformátumon alapul, és lehetővé teszi, hogy az összes osztályodat egyetlen JAR fájlba csomagold ahelyett, hogy az ügyfélnek 28 osztályfájlt adnál át. Ha ismered a UNIX tar parancsát, a jar eszközparancsait is fel fogod ismerni. (Megjegyzés: amikor azt látod, hogy JAR – csupa nagybetűvel –, akkor az archívum *fájlját* értjük alatta, míg ha kisbetűvel szerepel, akkor a *jar* segédprogramot, amely a JAR fájlok létrehozására használatos.)

A kérdés az, hogy mit *csinál* az ügyfél a befőttesüveggel (a JAR fájllal)? Hogyan veszi rá, hogy fusson? Nos, úgy, hogy ***végrehajthatóvá*** teszed a JAR fájlt.

A végrehajtható JAR fájl azt jelenti, hogy a végfelhasználónak nem kell kinyernie az osztályfájlokat, mielőtt futtathatná a programot. A felhasználó nyugodtan futtathatja a programot, miközben az osztályfájlok továbbra is a JAR fájlban találhatók. Ennek az a trükkje, hogy létre kell hozni egy úgynevezett ***jegyzékfájlt*** (manifest), ami szintén bekerül a befőttesüvegbe, és információkat tárol a JAR-ban található fájlokról. Ahhoz, hogy a JAR végrehajtható legyen, a jegyzéknek el kell árulnia a JVM-nek, hogy *melyik osztályban található a main() tagfüggvény!*



Végrehajtható JAR fájl készítése

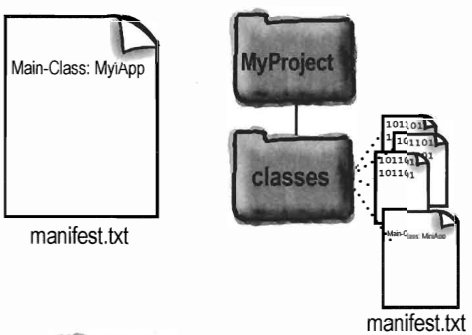
1. **Gondoskodj róla, hogy minden osztályfájlod a classes könyvtárban legyen**
   Ezen a következő oldalakon még finomítunk, de egyelőre elég, ha minden osztályfájl a „classes" nevű könyvtárban várakozik.

2. **Hozz létre egy manifest.txt fájlt, amely leírja, hogy melyik osztályban található a main() tagfüggvény**
   Hozz létre egy manifest.txt nevű szövegfájlt, amelyben egyetlen sor található:

   ```
   Main-Class: MyApp
   ```

   A végére ne írd oda a .class kiterjesztést!

   Nyomd le az ENTER/RETURN billentyűt a Main-Class sor után, különben a jegyzék esetleg nem fog megfelelően működni. Tedd a jegyzékfájlt a „classes" könyvtárba.

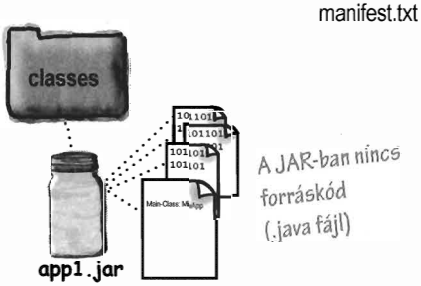


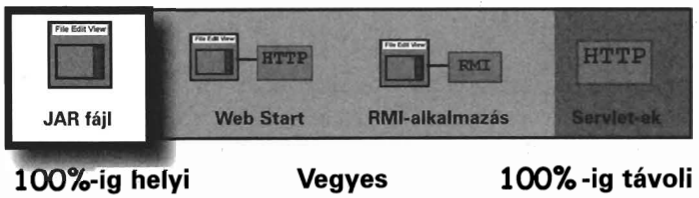
3. **Futtasd a jar segédprogramot a JAR fájl létrehozásához, amelybe minden belekerül, ami a classes könyvtárban található, a jegyzékfájllal együtt**

   ```
   %cd MiniProject/classes
   %jar -cvmf manifest.txt app1.jar *.class
   ```

   VAGY

   ```
   %jar -cvmf manifest.txt app1.jar MyApp.class
   ```

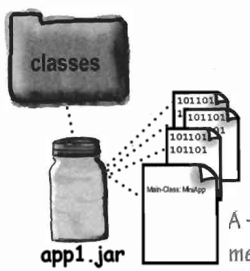




A 100%-ig helyi Java-alkalmazások többségét végrehajtható JAR fájlok formájában teszik közzé.

A JAR futtatása (végrehajtása)

A Java (a JVM) képes betölteni egy osztályt egy JAR fájlból, és meghívni annak a main() tagfüggvényét. Valójában a teljes alkalmazás a befőttesüvegben *maradhat*. Ha a kerekek egyszer forogni kezdtek (vagyis a main() tagfüggvény elindult), a JVM-et már nem érdekli, hogy *honnan* származnak az osztályok, amíg meg tudja találni azokat. Az egyik hely pedig, ahol a JVM körülnéz, az osztályelérési útvonalon található JAR fájlok belseje. Ha a JVM *lát* egy JAR-t, akkor bele is *néz*, amikor meg kell keresnie és be kell töltenie egy osztályt.




```
%cd MyProject/classes

%java -jar app1.jar
```

A JVM-nek „látnia" kell a JAR-t, ezért annak az osztályelérési útvonalon kell lennie. Egy JAR-t úgy tehetsz a legegyszerűbben láthatóvá, hogy munkakönyvtárként azt a helyet adod meg, ahol a JAR található.

A –jar kapcsoló azt közli a JVM-mel, hogy nem osztályt, hanem befőttesüveget adsz át neki.

A JVM ebben a JAR-ban keresi a Main-Class bejegyzést tartalmazó jegyzékfájlt. Ha nem találja, futásidejű kivétel lép fel.

Az operációs rendszered beállításaitól függően az is lehet, hogy elég duplán kattintani a JAR fájlra, hogy elindítsd. Ez a Windows és a Mac OS X legtöbb változatára igaz. Az indításhoz egyébként általában ki kell jelölnöd a JAR-t, és közölni az operációs rendszerrel, hogy melyik programmal nyissa meg (Open With, Társítás – vagy ami a rendszereden ennek a parancsnak a megfelelője).

Nincsenek hülye kérdések

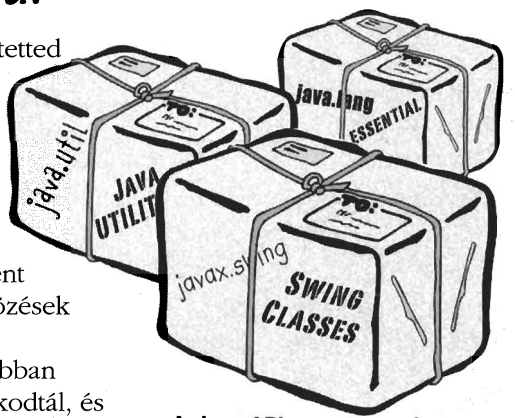
K: Nem lehet egyszerűen befőttesüvegbe tenni egy egész könyvtárat?

V: A JVM belenéz a befőttesüvegbe, és arra számít, hogy amit keres, az *rögtön ott* van. Tehát nem kezd el további könyvtárakba leásni, hacsak az osztály nem egy csomag része, de a JVM *még ekkor* is csak azokat a könyvtárakat nézi végig, amelyek illeszkednek a csomagutasításra.

K: Mit is mondtál az előbb?

V: Az osztályfájljaidat nem teheted tetszőleges könyvtárba, hogy aztán az egészet egyben gyömöszöld bele egy befőttesüvegbe. Ha viszont az osztályaid csomagokhoz tartoznak, a teljes csomagkönyvtár-szerkezetből JAR fájlt készíthetsz – sőt ez *kötelező* is. Lazíts, a következő oldalon elmagyarázzuk ezt az egészet.

Tedd az osztályaidat csomagokba!

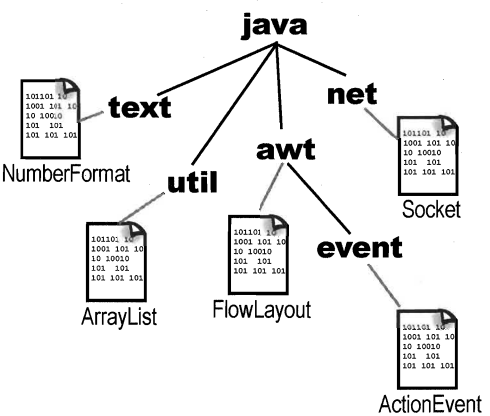


Írtál néhány szépen újrahasznosítható osztályfájlt, és közzétetted a belső fejlesztési könyvtárban, hogy a többi programozó felhasználhassa őket. Ahogy annak dicsfényében sütkérezel, hogy (szerény véleményed szerint) éppen most készítetted el a valaha volt legjobb objektumközpontú kódokat, egyszercsak befut egy telefonhívás – méghozzá egy elég ideges. Két osztályod neve megegyezik azoknak az osztályoknak a nevével, amelyeket Fred az imént helyezett a könyvtárba, ezért elszabadul a pokol: a névütközések és az egyértelműség hiánya térdre kényszeríti a fejlesztést.
És mindez azért, mert nem használtál csomagokat! Pontosabban használtál, hiszen a Java API meglevő osztályaira is támaszkodtál, és azok csomagokban találhatók, a saját osztályaidat azonban nem tetted csomagokba, ami a Valóságban Nagyon Csúnya Dolog.
Az előző oldalakon látott rendszerezést tehát egy kissé módosítani fogjuk, hogy az osztályok egy csomagba kerüljenek, a teljes csomag pedig egy JAR befőttesövegbe. Nagyon ügyelj a részletekre, mert a legkisebb eltérés is megakadályozhatja a kód sikeres lefordítását vagy futtatását!

A Java API csomagszerkezete a következőkhöz:
java.text.NumberFormat
java.util.ArrayList
java.awt.FlowLayout
java.awt.event.ActionEvent
java.net.Socket

A csomagok megakadályozzák az osztálynevek ütközését

Bár a csomagok nem csak a névütközések megakadályozását szolgálják, ez nagyon fontos előnyük. Írhatsz például egy Customer, egy Account és egy ShoppingCart nevű osztályt – anélkül, hogy tudnál róla, hogy az e-kereskedelmi üzletben érdekelt cégeknél dolgozó fejlesztőknek valószínűleg a fele készített már ilyen nevű osztályokat. Márpedig az objektumközpontúság világában ez veszélyes.
Az objektumközpontú tervezés egyik fő célja az, hogy a programozók újrahasznosítható összetevőket készíthessenek, márpedig ehhez hozzá tartozik, hogy a fejlesztők különböző forrásokból származó elemeket illeszthessenek össze, hogy valami újat alkossanak belőlük. Az összetevőidnek tehát „össze kell passzolniuk" a többi elemmel, beleértve azokat is, amelyeket nem te írtál, sőt azokkal is, amelyeknek a létezéséről nem is tudsz.
Emlékezhetsz rá, hogy a 6. fejezetben azt mondtuk, hogy a csomagnév olyan, mint az osztály teljes neve, hivatalos elnevezéssel a *teljesen minősített név* (fully-qualified name). Az ArrayList osztály teljes neve valójában ***java.util.ArrayList***, a JButton-é ***javax.swing.JButton***, a Socket-é pedig ***java.net.Socket***. Észreveheted, hogy az említett osztályok közül kettőnek – ArrayList és Socket – a nevében ugyanúgy a *java* az első tag, vagyis a teljesen minősített nevük egyaránt úgy kezdődik, hogy „java". A csomagok hierarchiába rendeződnek, így az osztályaidat ennek megfelelően kell rendszerezned.



Szerinted mire hasonlít ez az ábra? Nem pont úgy néz ki, mint egy könyvtár-hierarchia?

...így aztán a kvantum-főző-osztályom neveként végül a foo.bar.Heisenberg mellett döntöttem.

Tényleg? Én pont ugyanerre gondoltam, amikor a szubatomi vasalóosztályomnak kerestem nevet! De így azt hiszem, másikat kell keresnem...



A csomagnevek ütközésének megakadályozása

Ha egy osztályt egy csomagba teszel, az csökkenti a más osztályokkal való névütközés esélyét, de mi akadályozza meg, hogy két programozó azonos *csomagnevekkel* álljon elő? Másként megfogalmazva, mi áll az útjába annak, hogy két programozónak egyaránt legyen egy Account nevű osztálya, és mindkettő egy shopping.customers nevű csomagba tegye az osztályt? Ebben az esetben természetesen mindkét osztálynak *azonos* lesz a neve:

shopping.customers.Account

A Sun egy olyan csomagelnevezési egyezményt ajánl, ami jelentősen csökkenti ezt a kockázatot: azt, hogy minden osztály neve elé tegyük ki előtagként a tartománynevünket megfordítva. Ne feledd, hogy a tartománynevek garantáltan egyediek: két embert hívhatnak egyformán Bartholomew Simpsonnak, de két különböző tartománynak nem lehet egyaránt doh.com a neve.

A csomagok képesek megakadályozni a névütközéseket, de csak akkor, ha mindig olyan csomagnevet választasz, ami biztosan egyedi. Ezt a legegyszerűbben úgy biztosíthatod, ha a csomagokat a megfordított tartománynéveddel, mint előtaggal látod el.

com.headfirstbooks.Book

csomagnév

osztálynév

Fordított tartományt tartalmazó csomagnevek

com.headfirstjava.projects.Chart ← Az osztály neve mindig nagybetűvel kezdődik.

A csomag nevét a tartományneved megfordításával kezdd, majd egy ponttal elválasztva add hozzá a szervezési struktúrát.

A projects.Chart gyakori név lehet, de a com.headfirstjava előtagnak köszönhetően csak a házon belüli fejlesztők miatt kell aggódnod.

Így tehetsz egy osztályt egy csomagba:

Az osztályaidat kötelező olyan könyvtárszerkezetben elhelyezned, ami megegyezik a csomagok hierarchiájával.

1 Válassz egy csomagnevet

A példában a com.headfirstjava tartománynevet fogjuk használni. Az osztály neve PackageExercise („csomaggyakorlat") lesz, tehát teljesen minősített névként ezt kapjuk:
com.headfirstjava.PackageExercise.

2 Helyezz el egy package utasítást az osztályban

A package („csomag") utasításnak az első utasításnak kell lennie a forráskódfájlban, minden import utasítást megelőzve. Egy forráskódfájlban csak egy package utasítás lehet, tehát **egy forrás minden osztályának ugyanabba a csomagba kell kerülnie**. Ebbe a belső osztályok is beleértendők.

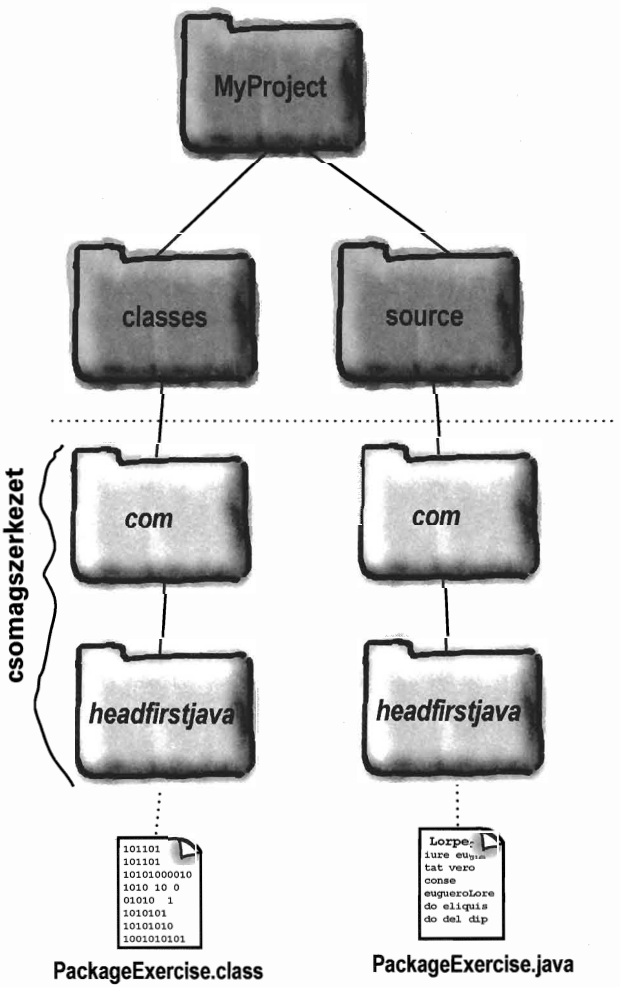
```
package com.headfirstjava;

import javax.swing.*;

public class PackageExercise {
    // ide jön az egész életedet
}   // megváltoztató kód
```

3 Állíts fel egy megfelelő könyvtárszerkezetet

Nem elég csak ***mondani***, hogy az osztály egy csomagban található, és egyszerűen elhelyezni egy package utasítást a kódban. Az osztály csak akkor kerül *valóban* egy csomagba, amikor az osztályt elhelyezed egy megfelelő könyvtárszerkezetben. Ha a teljesen minősített osztálynév tehát com.headfirstjava.PackageExercise, akkor a PackageExercise forráskódját egy **headfirstjava** nevű könyvtárba ***kell*** tenned, annak pedig egy **com** nevű könyvtárban ***kell*** lennie.
A fordítás *enélkül* is végrehajtható, de olyan problémákat idéz elő, hogy hidd el nekünk, nem éri meg. A forráskódot tartsd olyan könyvtárszerkezetben, ami megegyezik a csomagszerkezettel, így később rengeteg fejfájástól megkímélheted magad.



Állíts fel egy megfelelő könyvtárszerkezetet mind a source, mind a classes fa számára.

Fordítás és futtatás csomagokkal

Amikor egy osztályod egy csomagban található, kicsit furfangosabb módon lehet lefordítani és futtatni. A lényeg az, hogy a fordítónak és a JVM-nek egyaránt meg kell tudnia találni az osztályt és minden más osztályt is, amelyekre az támaszkodik. A központi API osztályainak esetében ez soha nem jelent gondot, hiszen a Java mindig tudja, hol vannak a saját dolgai. A saját osztályaid esetében azonban nem fordíthatsz ugyanabból a könyvtárból, ahol a forrásfájljaid is találhatók – ez egyszerűen nem működik (de legalábbis nem *megbízhatóan*). Azt viszont garantáljuk, hogy ha az ezen az oldalon leírt lépéseket követed, sikerrel fogsz járni. Másképp is megoldható a dolog, de mi ezt találtuk a legmegbízhatóbb és a legkönnyebben követhető megoldásnak.

Fordítás a –d (directory) kapcsolóval

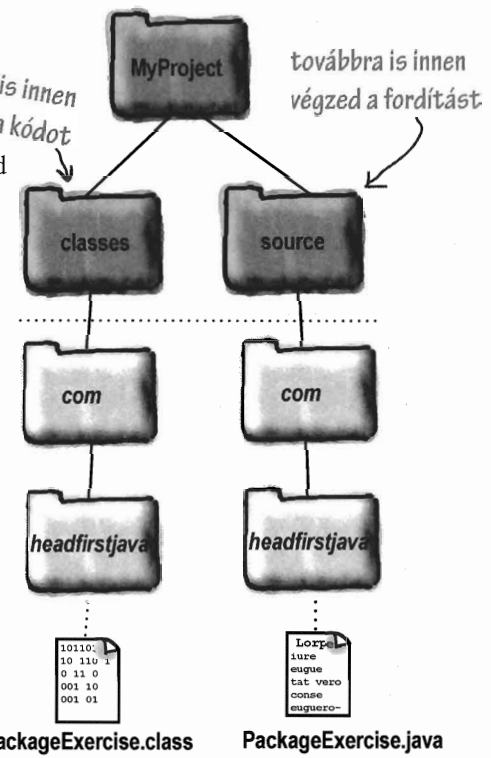
```
%cd MyProject/source
```

Maradj a forráskönyvtárban! NE válts (cd) a .java fájlt tartalmazó könyvtárra!

```
%javac -d ../classes com/headfirstjava/PackageExercise.java
```

Ez arra utasítja a fordítót, hogy a lefordított kódot (az osztályfájlokat) a „classes" könyvtárba tegye, a megfelelő csomagszerkezeten belül!!! Igen, ismeri.

Most meg kell adnod a tényleges forrásfájl ELÉRÉSI ÚTJÁT.



Ha a com.headfirstjava csomagban található összes .java fájlt le akarod fordíttatni, akkor ezt kell írnod:

```
%javac -d ../classes com/headfirstjava/*.java
```

Ez minden forrásfájlt (.java) lefordít ebben a könyvtárban.

A kód futtatása

```
%cd MyProject/classes
```

A program futtatása a „classes" könyvtárból

```
%java com.headfirstjava.PackageExercise
```

KÖTELEZŐ a teljesen minősített osztálynevet megadnod! A JVM felismeri azt, és nyomban belenéz az aktuális könyvtárba (classes), ahol elvárja, hogy találjon egy com nevű könyvtárat, abban egy headfirstjava nevűt, abban pedig az osztályt. Ha az osztály a „com" vagy netán a „classes" könyvtárban található, a dolog nem fog működni!

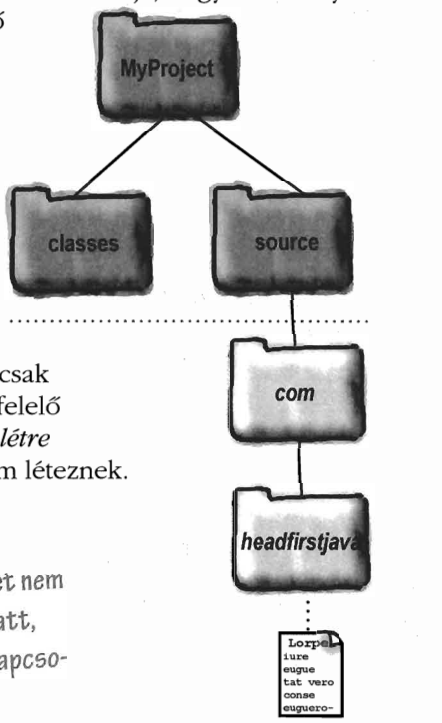
A –d kapcsoló még klasszabb, mint ahogy korábban mondtuk

A **-d** kapcsolóval fordítani csodálatos, mert nem csak azt teszi lehetővé, hogy a lefordított osztályt másik könyvtárba irányítsd, mint ahol a forrásfájlja található, hanem azt is tudja, hogy az osztályt az őt tartalmazó csomagnak megfelelő könyvtárszerkezetben hol kell elhelyezni.

És ez még nem minden!

Tegyük fel, hogy felállítottál egy tetszetős könyvtárszerkezetet a forráskód számára, de még nincs ehhez illeszkedő könyvtárszerkezeted a classes könyvtár alatt. Nem gond!

Ha a –d kapcsolóval fordítasz, a fordítót arra utasítod, hogy ne csak *elhelyezze* az osztályaidat a megfelelő könyvtárfában, hanem *hozza is létre* a könyvtárakat, ha azok még nem léteznek.



Ha a csomagkönyvtár-szerkezet nem létezik a „classes" könyvtár alatt, a fordító felépíti azt, ha a –d kapcsolót használod.

Így tehát nem kell fizikailag létrehoznod a „classes" gyökérkönyvtár alatti könyvtárakat, sőt ha a fordítóra bízod a létrehozásukat, az elgépelés esélyét is megszünteted.

A –d kapcsoló azt mondja a fordítóprogramnak, hogy „helyezd el az osztályt a csomagkönyvtár-szerkezetében, gyökérkönyvtárként a –d után megadott osztályt használva, de ha a könyvtárak nem találhatók, először hozd létre őket, és ez után tedd az osztályt a megfelelő helyre!".

Nincsenek

hülye kérdések

K: Megpróbáltam a cd paranccsal a fő osztályom könyvtárára váltani, de a JVM azt mondja, hogy nem találja az osztályt! Pedig OTT van az aktuális könyvtárban!

V: Ha egy osztályt egy csomagba tettél, többé nem szólíthatod a „rövid" nevén. A parancssorban meg KELL adnod a teljesen minősített nevét annak az osztálynak, amelyiknek a main() tagfüggvényét futtatni szeretnéd. Mivel azonban a teljesen minősített név tartalmazza a *csomagszerkezetet* is, a Java ragaszkodik hozzá, hogy az osztály ennek megfelelő *könyvtárszerkezetben* legyen. Ha tehát a parancssorban ezt mondod:

```
%java com.foo.Book
```

...akkor a JVM az aktuális könyvtárban (és az elérési útja többi részében) egy „com" nevű könyvtárat fog keresni. ***Book nevű osztályt nem keres, amíg meg nem találta a „com" könyvtárat, benne egy „foo" nevű könyvtárral.*** A JVM csak ekkor fogadja el, hogy megtalálta a keresett Book osztályt. Ha bárhol máshol akad Book osztályra, feltételezi, hogy az osztály nem megfelelő szerkezetben lett elhelyezve – akkor is, ha ez az igazság! A JVM nem kúszik feljebb például a könyvtárfában azzal, hogy „látom, felettem van egy com nevű könyvtár, úgyhogy biztos ez a megfelelő csomag...".

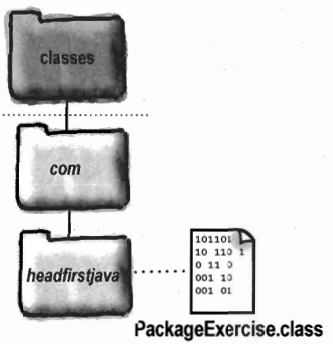
Végrehajtható JAR fájl készítése



Ha egy osztályodat csomagba rakod, a csomag könyvtárszerkezetének egy befőttesüveg, vagyis egy JAR fájl belsejében kell lennie. Az osztályaidat nem dobálhatod be csak úgy a befőttesüvegbe, ahogy a csomagok használata előtt tetted. Ezenkívül arra is ügyelned kell, hogy a csomag felett levő egyetlen könyvtár se kerüljön bele a JAR-ba. A csomag első könyvtárának (ez általában a „com" nevet viseli) az első könyvtárnak kell lennie a JAR fájl belsejében. Ha véletlenül beemeled a csomag *feletti* (vagyis a „classes") könyvtárat is, a JAR nem fog megfelelően működni.

Végrehajtható JAR fájl készítése

1 Gondoskodj róla, hogy minden osztályfájlod a megfelelő csomagszerkezetben legyen, a classes könyvtár alatt.

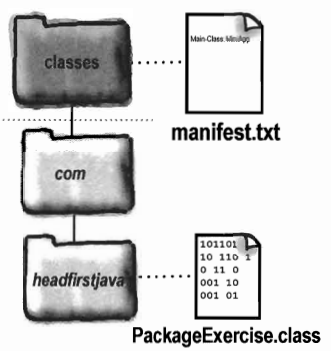


2 Hozz létre egy manifest.txt fájlt, amely leírja, hogy melyik osztályban található a main() tagfüggvény, és ügyelj rá, hogy a teljesen minősített osztálynevet használd.

Hozz létre egy manifest.txt nevű szövegfájlt, amelyben egyetlen sor található:

```
Main-Class: com.headfirstjava.PackageExercise
```

Tedd a jegyzékfájlt a „classes" könyvtárba.



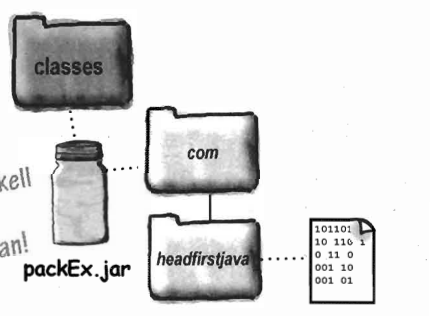
3 Futtasd a jar segédprogramot a JAR fájl létrehozásához, amelybe a csomag minden könyvtára belekerül, a jegyzékfájllal együtt.

Az egyetlen dolog, amit be kell emelned, a „com" könyvtár, és a teljes csomag (valamennyi osztállyal együtt) a befőttesüvegbe kerül.

```
%cd MyProject/classes

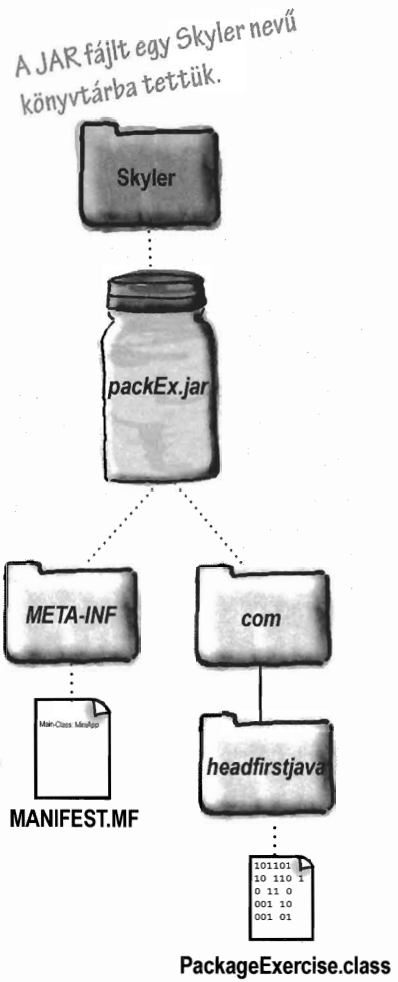
%jar -cvmf  manifest.txt  packEx.jar  com
```

Csak a com könyvtárat kell megadnod, és mindent megkapsz, ami benne van!



És hová kerül a jegyzékfájl?

Miért nem nézünk bele a JAR fájlba, és derítjük ki? A parancssorból a jar segédprogram többre is képes annál, mint hogy csak létrehozzon és futtasson egy JAR-t. A befőttessüveg tartalmát például ki is nyerheted (ami olyan, mint az „unzip" vagy az „untar").
Tegyük fel, hogy a packEx.jar fájlt egy Skyler nevű könyvtárba tetted.



A jar segédprogram kiíró és kibontó parancsai

1 Írasd ki a JAR fájl tartalmát

```
% jar -tf packEx.jar
```

A –tf a „Table File" (tartalomjegyzék-fájl) rövidítése, vagyis ez a parancs annyit tesz, hogy „mutasd meg nekem a JAR fájl tartalomjegyzékét".

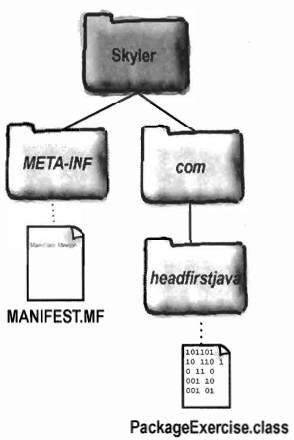
```
File Edit Window Help Pickle
% cd Skyler
% jar -tf packEx.jar
META-INF/
META-INF/MANIFEST.MF
com/
com/headfirstjava/
com/headfirstjava/
PackageExercise.class
```

A jar segédprogram automatikusan létrehoz egy META-INF nevű könyvtárat, és ebbe teszi a jegyzékfájlt.

2 Nyerd ki a JAR fájl tartalmát („bontsd ki a befőttet")

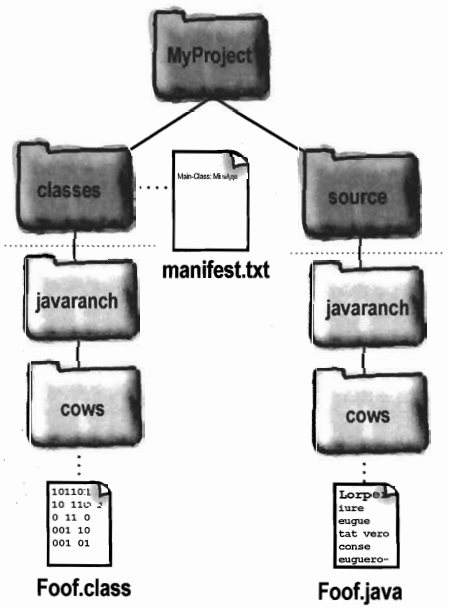
```
% cd Skyler
% jar -xf packEx.jar
```

Az –xf az „Extract File" (fájl kibontása) rövidítése, és ugyanúgy működik, mint az unzip vagy az untar. Ha kibontod a packEx.jar fájlt, az aktuális könyvtárban a META-INF, illetve a com könyvtárat fogod látni.



A META-INF a „meta information" rövidítése. A jar segédprogram létrehozza a META-INF könyvtárat, valamint a MANIFEST.MF fájlt, illetve fogja a te jegyzékfájlod tartalmát, és beleteszi a MANIFEST.MF fájlba. A te manifest fájlod tehát nem kerül a JAR-ba, a tartalma azonban a „valódi" jegyzékfájlt (MANIFEST.MF) gazdagítja.

Hegyezd ki a ceruzád!



A képen látható csomag-, illetve könyvtárszerkezetet alapul véve állapítsd meg, hogy mit kell beírnod a parancssorba a fordításhoz, futtatáshoz, illetve a JAR fájl létrehozásához és végrehajtásához. Tételezd fel, hogy a szabványt követjük, ahol a csomagkönyvtár-szerkezet pontosan a *source* és *classes* könyvtárak alatt kezdődik. Más szavakkal, a *source* és *classes* könyvtárak nem képezik részét a csomagnak.

Fordítás:

```
%cd source
%javac _____
```

Futtatás:

```
%cd _____

%java _____
```

JAR létrehozása

```
%cd _____

% _____
```

JAR végrehajtása

```
%cd _____

% _____
```

Bónusz kérdés: mi nem stimmel a csomag nevével?

Nincsenek

hülye kérdések

K: Mi történik, ha megpróbálok futtatni egy végrehajtható JAR-t, de a végfelhasználónál nincs telepítve a Java?

V: Semmi sem fog elindulni, mivel JVM nélkül semmilyen Java-kód nem futtatható. A végfelhasználónak tehát muszáj telepítenie a Javát.

K: Hogyan érhetem el, hogy a Java telepítve legyen a végfelhasználó számítógépére?

V: Ideális esetben létrehozhatsz egy egyedi telepítőt, és mellékelheted az alkalmazásodhoz. Számos cég kínál telepítőprogramokat, az egyszerűtől a rendkívül sokoldalúig. Egy telepítőprogram például észlelheti, hogy a végfelhasználó rendelkezik-e a Java megfelelő változatával, és ha nem, telepítheti és beállíthatja a Javát, mielőtt telepítené az alkalmazásodat.
Az Installshield, az InstallAnywhere és a DeployDirector egyaránt kínál Java-telepítő megoldásokat.
Egyes telepítőprogramokban az is klassz, hogy akár telepítő CD-ROM-ot is készíthetsz, amelyen minden jelentősebb Java-platformhoz helyet kaphat egy-egy telepítő – egy CD mind felett. Ha a felhasználó például Solarist futtat, a Java Solarisra írt változata fog települni, míg Windowson a windowsos változat, és így tovább. Ha van rá elég pénzed, ez messze a legegyszerűbb módja annak, hogy a végfelhasználók gépén telepítsd és beállítsd a megfelelő Java-változatot.

Újratöltve

- Úgy rendszerezd a programodat, hogy a forráskód és az osztályfájlok ne ugyanabban a könyvtárban legyenek.
- Az egyik szabványos rendszerezési szerkezet az, hogy létrehozol egy *project* könyvtárat, azon belül pedig egy *source* (forrás), illetve egy *classes* (osztályok) nevű alkönyvtárat.
- Az osztályaid csomagokba szervezése megakadályozza a más osztályokkal való névütközést, ha az osztályaid neve elé előtagként odaírod a tartománynevedet, megfordítva.
- Ha egy osztályt csomagba szeretnél tenni, a forráskódfájl elején, minden import utasítás előtt el kell helyezned egy package utasítást:

```
package com.wickedlysmart;
```

- Egy osztály csak akkor kerül csomagba, ha a *csomagszerkezettel pontosan megegyező könyvtárszerkezetben található.* A com.wickedlysmart.Foo osztálynak például a *wickedlysmart* nevű könyvtárban kell helyet kapnia, a *com* könyvtár alatt.
- Ahhoz, hogy a lefordított osztály a megfelelő csomagkönyvtárszerkezetbe kerüljön a *classes* könyvtár alatt, használd a **-d** fordítói kapcsolót:

```
% cd source
% javac –d ../classes com/wickedlysmart/Foo.java
```

- A kód futtatásához válts a classes könyvtárra, és add meg az osztályod teljesen minősített nevét:

```
% cd classes
% java com.wickedlysmart.Foo
```

- Az osztályaidat „befőttesüvegekbe", vagyis JAR (Java ARchive) fájlokba is csomagolhatod. A JAR a pkzip formátumon alapul.
- Végrehajtható JAR fájlt úgy hozhatsz létre, hogy a befőttesüvegben elhelyezel egy jegyzékfájlt (manifest), amely leírja, hogy melyik osztályban található a main() tagfüggvény. A jegyzékfájl létrehozásához készíts egy szövegfájlt, amelyben például a következő bejegyzés szerepelhet:

```
Main-Class: com.wickedlysmart.Foo
```

- Ne felejtsd el leütni az Enter/Return billentyűt, miután beírtad a Main-Class sort, különben a jegyzékfájl nem fog működni.
- Egy JAR fájlt ehhez hasonlóan hozhatsz létre:

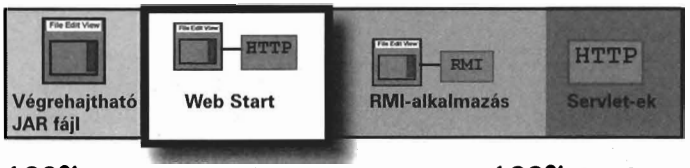
```
jar -cvfm manifest.txt MyJar.jar com
```

- A teljes csomagkönyvtár-szerkezetnek (és *csak* a csomagnak megfelelő könyvtáraknak) közvetlenül a JAR fájl belsejében kell lenniük.
- Egy végrehajtható JAR fájlt így futtathatsz:

```
java -jar MyJar.jar
```



A végrehajtható JAR fájlok klasszak, de nem lenne meseszép, ha lenne rá mód, hogy sokoldalú, önálló, grafikus ügyfélfelületet készítsünk, amit a Weben keresztül tehetünk közzé? Így nem kellene CD-ROM-okat gyártatni és terjeszteni. És nem lenne csodás, ha a program képes lenne önműködően frissíteni magát, csak azokat a részeket kicserélve, amelyek megváltoztak? Az ügyfelek így mindig naprakészek lennének, és soha nem kellene aggódnunk amiatt, hogy miként juttatjuk el az új változatot a felhasználókhoz. Ez persze csak álom...



Java Web Start

A Java Web Start (JWS) használatakor az alkalmazásodat egy webböngésző indítja el (*web start*, fogod?), de (*majdnem*) önálló alkalmazásként fut, a böngésző korlátai nélkül. Miután letöltődött a végfelhasználó számítógépére (ami akkor következik be, amikor a felhasználó először kattint a böngészőben a letöltést elindító hivatkozásra), az alkalmazás *ott marad.*

A Java Web Start többek között egy apró Java-program, ami az ügyfélgépen lakozik, és a böngészők bővítményeihez (plug-in) hasonlóan működik (úgy, mint ahogy mondjuk az Adobe Acrobat Reader, ami akkor indul el, amikor a böngésző egy .pdf fájl kap). Ezt a Java-programot **Java Web Start-segédprogramnak** („helper app") hívják, és a legfontosabb feladata az, hogy kezelje a *te* JWS-alkalmazásaid letöltését, frissítését és elindítását (végrehajtását).

Amikor a JWS letölti az alkalmazásodat (egy végrehajtható JAR fájlt), meghívja annak main() tagfüggvényét. Ezt követően a végfelhasználó *anélkül* indíthatja el az alkalmazást a JWS-segédprogramból, hogy vissza kellene mennie a weboldalon található hivatkozáshoz.

De ez még nem minden. A JWS-ben az a lenyűgöző, hogy képes észlelni, ha az alkalmazásnak akár csak egy apró része (mondjuk egytelen osztályfájlja) is megváltozott a kiszolgálón, és – a végfelhasználó bármilyen közreműködése nélkül – letölteni és beépíteni a frissített kódot.

Egy probléma persze megoldásra vár: hogyan *jut hozzá* a végfelhasználó a Javához és a Java Web Starthoz? Mindkettőre szüksége van: a Javára az alkalmazás futtatásához, a Java Web Startra pedig (amely maga is egy kis méretű Java-alkalmazás) az alkalmazás letöltésének és elindításának kezeléséhez. Szerencsére azonban *ezt* a kérdést is megoldották. Beállíthatod, hogy ha a végfelhasználó nem rendelkezik a JWS-sel, akkor letöltheti azt a Suntól, ha pedig *megvan* neki, de a Java-változata elavult (mert a JWS-alkalmazásban egy konkrét Java-változatot írtál elő), akkor a Java 2 Standard Edition töltődjön le a végfelhasználó számítógépére.

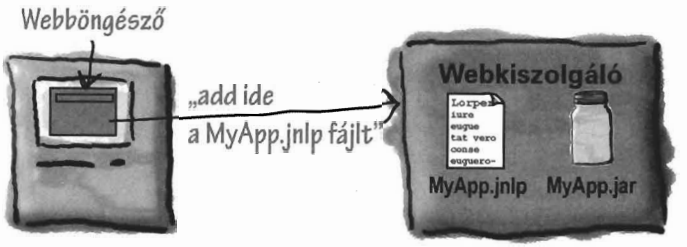
Ami pedig a legjobb az egészben: a Java Web Start használata egyszerű. Egy JWS-alkalmazás lényegében ugyanúgy érhető el, mint bármely más webes erőforrás, például egy hagyományos HTML-oldal vagy egy JPEG-kép: csak készítesz egy weboldalt (HTML-oldalt) egy hivatkozással, ami a JWS-alkalmazásodra mutat, és már mehet is.

Végeredményben tehát a JWS-alkalmazásod nem más, mint egy végrehajtható JAR fájl, amelyet a végfelhasználók letölthetnek a Webről.

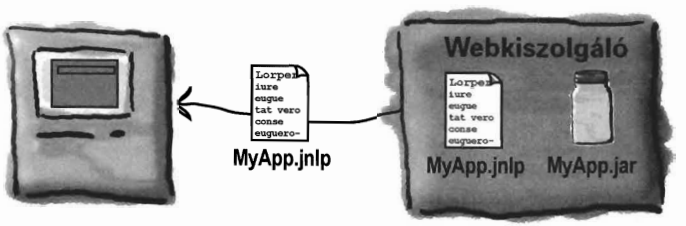
A végfelhasználók a Java Web Start-alkalmazásokat egy weboldalon található hivatkozásra kattintva tölthetik le. Miután azonban az alkalmazás letöltődött, a böngészőn kívül fut, mint bármely más önálló Java-alkalmazás. Valójában egy Java Web Start-alkalmazás csupán egy végrehajtható JAR fájl, amelyet a Weben keresztül teszel közzé.

A Java Web Start működése

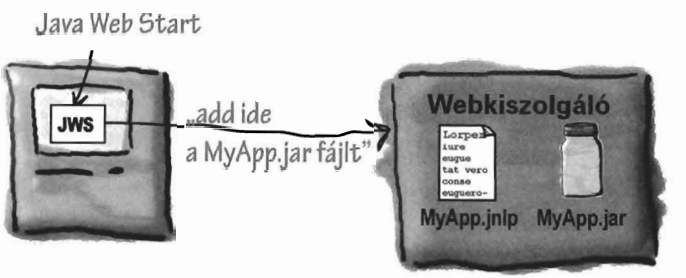
1. **Az ügyfél egy weboldalon található hivatkozásra kattint, ami a JWS-alkalmazásodra (egy .jnlp fájlra) mutat.**
   A webhivatkozás így fest:
   ```
   <a href="MyApp.jnlp">Click</a>
   ```



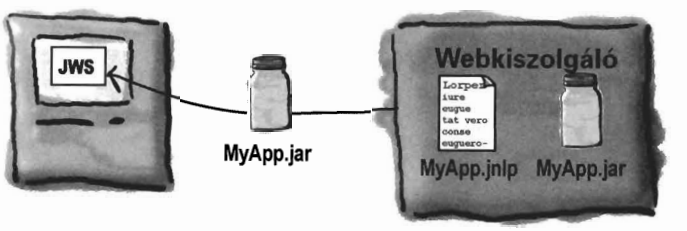
2. **A webkiszolgáló (HTTP) megkapja a kérelmet, és visszaküld egy .jnlp fájlt (tehát NEM a JAR-t).**
   A .jnlp fájl egy XML-dokumentum, amely megadja az alkalmazás végrehajtható JAR fájljának nevét.



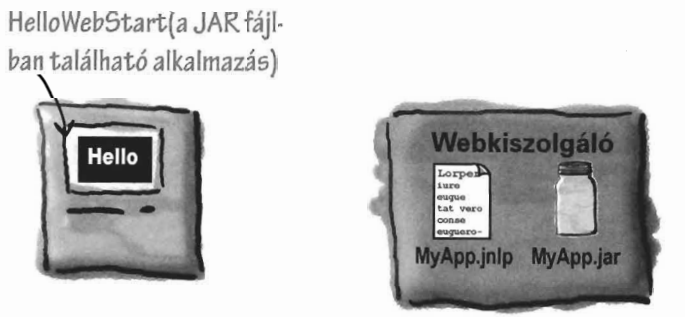
3. **A böngésző elindítja a Java Web Startot (egy apró „segédprogramot" az ügyfélen). A JWS-segédprogram elolvassa a .jnlp fájlt, és elkéri a kiszolgálótól a MyApp.jar fájlt.**



4. **A webkiszolgáló „kiszolgáltatja" a kért .jar fájlt.**



5. **A Java Web Start megkapja a JAR fájlt, és elindítja az alkalmazást a megadott main() tagfüggvény meghívásával.**
   Amikor a felhasználó legközelebb futtatni szeretné ezt az alkalmazást, megnyithatja a Java Web Start-ot, és onnan elindíthatja az alkalmazásodat, akár internetkapcsolat nélkül is.



A .jnlp fájl

Egy Java Web Start-alkalmazás készítéséhez szükséged van egy .jnlp (Java Network Launch Protocol) fájlra, amely leírja az alkalmazást. Ezt a fájlt olvassa be a JWS-alkalmazás, és ennek alapján találja meg a JAR fájlodat, illetve indítja el a programot (a JAR main() tagfüggvényének meghívásával). A .jnlp fájlok egyszerű XML-dokumentumok, amelyekbe különféle dolgokat írhatsz, de legalább az alábbiaknak szereplniük kell bennük:

```
<?xml version="1.0" encoding="utf-8"?>

<jnlp spec="0.2 1.0"
      codebase="http://127.0.0.1/~kathy"
      href="MyApp.jnlp">

   <information>
      <title>kathy App</title>
      <vendor>Wickedly Smart</vendor>
      <homepage href="index.html"/>
      <description>Head First WebStart demo</description>
      <icon href="kathys.gif"/>
      <offline-allowed/>
   </information>

   <resources>
      <j2se version="1.3+"/>
      <jar href="MyApp.jar"/>
   </resources>

   <application-desc main-class="HelloWebStart"/>
</jnlp>
```

A „codebase" elemben adhatod meg a Web Start-programod „gyökerét" a kiszolgálón. A programot itt a helyi állomáson (localhost) teszteljük, ezért a „127.0.0.1" helyi visszacsatoló címet használjuk. A mi internetes webkiszolgálónkon található Web Start-alkalmazások esetében ez azt jelenti, hogy „http://www.wickedlysmart.com".

Ez a .jnlp fájl helye a kódgyökérhez (codebase) viszonyítva. A példa azt mutatja, hogy a MyApp.jnlp a webkiszolgáló gyökérkönyvtárában érhető el, nem pedig valamelyik másik könyvtárba ágyazva.

Ügyelj rá, hogy ezek közül a címkék közül egyiket se hagyd ki, különben az alkalmazásod nem biztos, hogy helyesen fog működni! Az „information" címkéket a JWS-segédprogram használja, főként megjelenítésre, hogy a felhasználó kiválaszthassa, hogy melyik korábban letöltött alkalmazást szeretné elindítani.

Ez azt jelenti, hogy a felhasználó internetkapcsolat nélkül is elindíthatja a programodat. Ha a felhasználó kapcsolat nélkül dolgozik, az automatikus frissítés nem működik.

Ez azt mondja, hogy az alkalmazásodnak a Java 1.3-as vagy későbbi változatára van szüksége.

A végrehajtható JAR fájlod neve. Előfordulhat, hogy további JAR fájljaid is vannak, amelyek más olyan osztályokat - vagy akár hangokat és képeket tárolnak -, amelyeket az alkalmazásod használ.

Ez olyan, mint a jegyzékfájl Main-Class bejegyzése, és azt árulja el, hogy main() tagfüggvény a JAR fájl melyik osztályában található.

A Java Web Start-alkalmazások készítésének és közzétételének lépései

1 **Létrehozol egy végrehajtható JAR fájlt az alkalmazás számára.**



MyApp.jar

2 **Megírod a .jnlp fájlt.**



MyApp.jnlp

3 **A JAR és a .jnlp fájlt elhelyezed a webkiszolgálón.**



4 **Hozzáadsz egy új MIME-típust a webkiszolgálóhoz.**

```
application/x-java-jnlp-file
```

Ennek hatására a kiszolgáló a .jnlp fájlt helyes fejléccel küldi el, így amikor a böngésző megkapja a .jnlp fájlt, tudni fogja, hogy micsoda, és elindítja a JWS-segédprogramot.



5 **Létrehozol egy weboldalt, egy hivatkozással a .jnlp fájlra.**



MyJWSApp.html

```
<HTML>
  <BODY>
    <a href="MyApp2.jnlp">Launch My Application</a>
  </BODY>
</HTML>
```



Gyakorlat

Mi volt előbb?



Nézd meg az alábbi eseményeket, és tedd őket abba sorrendbe, ahogy egy JWS-alkalmazásban egymást követik!

A webböngésző elindítja a JWS-segédprogramot.

A webkiszolgáló elküldi a .jnlp fájlt a böngészőnek.

A felhasználó a webhivatkozásra kattint.

A webkiszolgáló elküldi a JAR fájlt a JWS-segédprogramnak.

A JWS-segédprogram elkéri a JAR fájlt.

A JWS-segédprogram meghívja a JAR main() tagfüggvényét.

A böngésző elkéri a .jnlp fájlt a webkiszolgálótól.

1.
2.
3.
4.
5.
6.
7.

Nincsenek

hülye kérdések

K: Miben különbözik a Java Web Start a kisalkalmazásoktól (applet-ektől)?

V: A kisalkalmazások nem élhetnek egy webböngészőn kívül: a Webről egy weboldal *részeként* töltődnek le, nem csupán egy weboldalon keresztül. Más szavakkal, a böngésző számára az applet ugyanolyan, mint egy JPEG-kép vagy bármely más erőforrás. A böngésző vagy egy Java-bővítményt, vagy a saját beépített Java-értelmezőjét (ez ma már sokkal kevésbé gyakori) használja a kisalkalmazás futtatására. A kisalkalmazások képességei korlátozottabbak, például nem tudnak önműködően frissülni, és mindig a böngészőből kell elindítani őket. A JWS-alkalmazásokat ezzel szemben a Webről történő letöltés után a felhasználónak nem muszáj egy böngésző segítségével elindítania, ha helyben szeretné használni őket. Ehelyett elég elindítani a JWS-segédprogramot, és azzal betölteni a korábban letöltött alkalmazást.

K: Milyen biztonsági korlátozások vonatkoznak a JWS-re?

V: A JWS-alkalmazások sok mindent nem tehetnek meg, például nem írhatnak a felhasználó merevlemezére, és nem is olvashatnak onnan. Ez alól csak a JWS saját API-ja kivétel, amely egyedi megnyitási és mentési párbeszédablakokat kínál, amelyeken keresztül az alkalmazás a felhasználó engedélyével mentheti és visszaolvashatja a saját fájljait, a merevlemez egy különleges, elzárt területéről.



Újratöltve

- A Java Web Start önálló ügyfélalkalmazások telepítését teszi lehetővé a Webről.
- A Java Web Startnak része egy „segédprogram", amelyet telepíteni kell az ügyfélre (a Javával együtt).
- A Java Web Start (JWS-) alkalmazások két részből állnak: egy végrehajtható JAR és egy .jnlp fájlból.
- A .jnlp fájlok egyszerű XML-dokumentumok, amelyek leírják az adott JWS-alkalmazást. A benne levő címkék adják meg a JAR fájl nevét és helyét, valamint a main() tagfüggvényt tartalmazó osztály nevét.
- Amikor egy böngésző egy .jnlp fájlt kap a kiszolgálótól (mert a felhasználó a .jnlp fájlhoz vezető hivatkozásra kattintott), a böngésző elindítja a JWS-segédprogramot.
- A JWS-segédprogram beolvassa a .jnlp fájlt, és elkéri a végrehajtható JAR fájlt a webkiszolgálótól.
- Amikor a JWS megkapja a JAR fájlt, meghívja a .jnlp fájlban megadott main() tagfüggvényt.



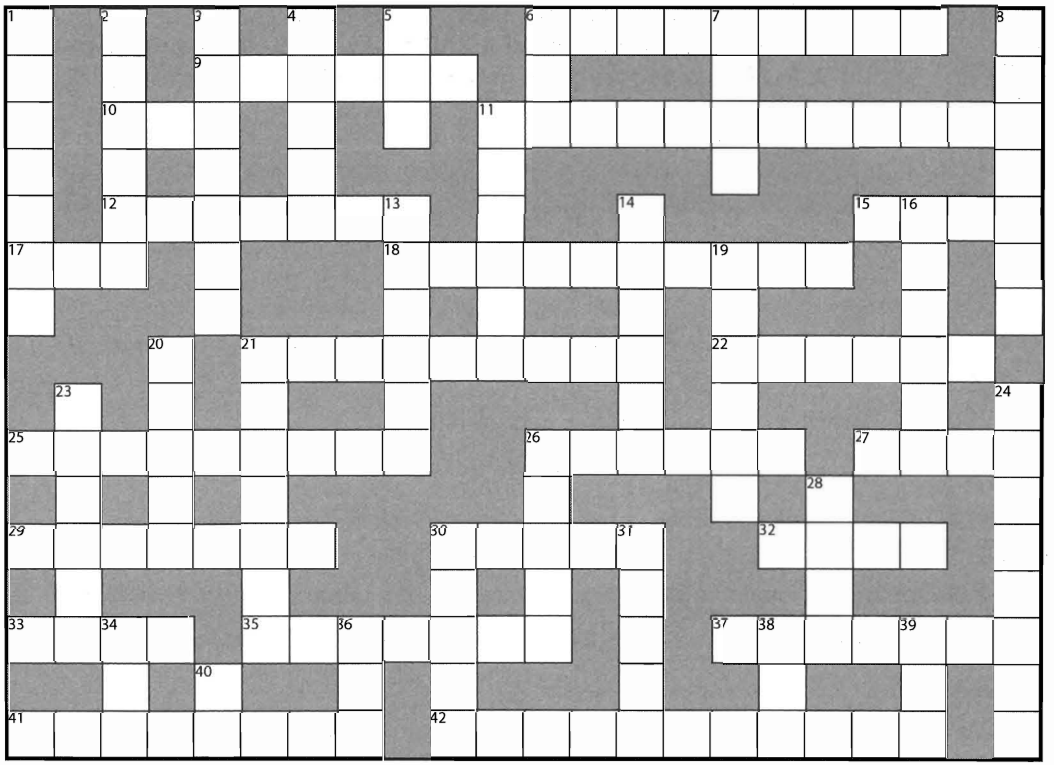
Ebben a fejezetben a csomagok, a közzététel és a JWS rejtelmei tárultak fel előtted. A feladatod az, hogy eldöntsd, hogy az alábbi állítások közül melyik igaz, és melyik hamis.

Igaz vagy hamis?

1. A Java-fordítónak van egy kapcsolója, a –d, amelynek a segítségével te határozhatod meg, hogy hová kerüljenek a .class fájljaid.
2. A JAR egy szabványos könyvtár – ide kell kerülnie a .class fájloknak.
3. Amikor Java-archívumot készítesz, létre kell hoznod egy jar.mf nevű fájlt.
4. A Java-archívumban található segédfájl adja meg, hogy melyik osztályban található a main() tagfüggvény.
5. A JAR fájlt ki kell csomagolni, mielőtt a JVM használatba vehetné a benne található osztályokat.
6. A Java-archívumokat a parancssorban az –arch kapcsolóval kell meghívni.
7. A csomagszerkezetet egy könyvtárhierarchia tükrözi.
8. A céged tartománynevének használata nem ajánlott a csomagok elnevezésénél.
9. Egy forrásfájlon belül a különböző osztályok különböző csomagokba is tartozhatnak.
10. Egy csomag osztályainak lefordításakor erősen javasolt a –p kapcsolót használni.
11. Egy csomag osztályainak lefordításakor a teljes névnek tükröznie kell a könyvtárfát.
12. A –d kapcsoló megfontolt használatával elkerülhető az osztályfa hibás megadása.
13. Egy csomagokat tartalmazó JAR kibontásakor létrejön egy meta-inf nevű könyvtár.
14. Egy csomagokat tartalmazó JAR kibontásakor létrejön egy manifest.mf nevű fájl.
15. A JWS-segédprogram mindig egy böngészővel együtt fut.
16. A JWS-alkalmazások helyes működéséhez egy .nlp (Network Launch Protocol) fájlra van szükség.
17. A JWS-alkalmazások main() tagfüggvényét az alkalmazás JAR fájlja adja meg.



Összefoglaló keresztrejtvény 7.0



Ebben a keresztrejtvényben bármi szerepelhet, ami a könyvben előfordult!

Vízszintes

6. Nem tart örökké
9. Oszthatatlan (angolul)
10. Befőttesüveg
11. Kulcs kell hozzá
12. Összekapcsolt (I/O) (angolul)
15. Eltesz
17. Betokozott olvasó
18. Az ilyen fájlokat teszed közzé (angolul)
21. Megvalósít, angolul
22. I/O-szita (angolul)
25. Mappa a lemezen (angolul)
26. Csatoló, angolul
27. Graphical ____ Interface
29. Java-csomag
30. Gyár
32. Egy ideig ismétlődik
33. Atomi * 8
35. Hogy olyan legyen, mint újkorában (angolul)
37. Határeset (angolul)
41. Ez mondja meg, hol kezdd
42. Körülfalaz (angolul)

Függőleges

1. Tolakodó ablakok (angolul)
2. A vágy titokzatos ____ (angoul)
3. Magára hagyott elem (angolul)
4. Elem, rész
5. Nem max
6. Légy bátor!
7. Elrendez
8. Összetevők Swing-szlengben (angolul)
11. I/O-csatorna (angolul)
13. Szervezetten közzétesz, telepít (angolul)
14. Nem egy pillanatra szól
16. Engedélyezve vagy megtagadva (angolul)
19. Hatékonyságnövelő (angolul)
20. Korai kilépés
21. Szokványos burkoló
23. Igen vagy nem (angolul)
24. Java-kabátok (angolul)
26. Nem viselkedés (angolul)
28. Egy csatoló lakhelye
30. I/O-takarítás
31. Rövid szunyókálás
34. Trigonometriai függvény
36. Beállító függvény
38. A JNLP formátuma
39. Végeredmény
40. Java-ág



Gyakorlat – Megoldás





1. A felhasználó a webhivatkozásra kattint.
2. A böngésző elkéri a .jnlp fájlt a webkiszolgálótól.
3. A webkiszolgáló elküldi a .jnlp fájlt a böngészőnek.
4. A webböngésző elindítja a JWS-segédprogramot.
5. A JWS-segédprogram elkéri a JAR fájlt.
6. A webkiszolgáló elküldi a JAR fájlt a JWS-segédprogramnak.
7. A JWS-segédprogram meghívja a JAR main() tagfüggvényét.

Igaz 1. A Java-fordítónak van egy kapcsolója, a –d, amelynek a segítségével te határozhatod meg, hogy hová kerüljenek a .class fájljaid.

Hamis 2. A JAR egy szabványos könyvtár – ide kell kerülnie a .class fájloknak.

Hamis 3. Amikor Java-archívumot készítesz, létre kell hoznod egy jar.mf nevű fájlt.

Igaz 4. A Java-archívumban található segédfájl adja meg, hogy melyik osztályban található a main() tagfüggvény.

Hamis 5. A JAR fájlt ki kell csomagolni, mielőtt a JVM használatba vehetné a benne található osztályokat.

Hamis 6. A Java-archívumokat a parancssorban az –arch kapcsolóval kell meghívni.

Igaz 7. A csomagszerkezetet egy könyvtárhierarchia tükrözi.

Hamis 8. A céged tartománynevének használata nem ajánlott a csomagok elnevezésénél.

Hamis 9. Egy forrásfájlon belül a különböző osztályok különböző csomagokba is tartozhatnak.

Hamis 10. Egy csomag osztályainak lefordításakor erősen javasolt a –p kapcsolót használni.

Igaz 11. Egy csomag osztályainak lefordításakor a teljes névnek tükröznie kell a könyvtárfát.

Igaz 12. A –d kapcsoló megfontolt használatával elkerülhető az osztályfa hibás megadása.

Igaz 13. Egy csomagokat tartalmazó JAR kibontásakor létrejön egy meta-inf nevű könyvtár.

Igaz 14. Egy csomagokat tartalmazó JAR kibontásakor létrejön egy manifest.mf nevű fájl.

Hamis 15. A JWS-segédprogram mindig egy böngészővel együtt fut.

Hamis 16. A JWS-alkalmazások helyes működéséhez egy .nlp (Network Launch Protocol) fájlra van szükség.

Hamis 17. A JWS-alkalmazások main() tagfüggvényét az alkalmazás JAR fájlja adja meg.



Összefoglaló keresztrejtvény 7.0

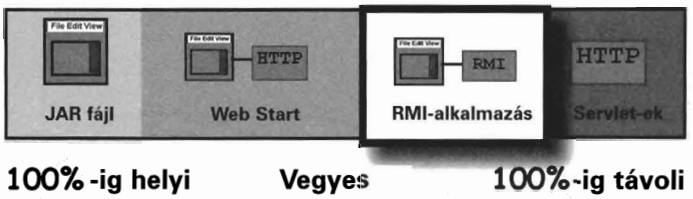
1 D		2 O		3 G		4 T		5 M			6 T	R	A	N	7 S	I	E	N	T		8 W
I		B		9 A	T	O	M	I	C		R				O						I
A		10 J	A	R		K		N		11 S	Y	N	C	H	R	O	N	I	Z	E	D
L		E		B		E				T					T						G
O		12 C	H	A	I	N	E	D		R			14 S						15 S	16 A	V	E
17 G	E	T		G				18 E	X	E	C	U	T	A	19 B	L	E		C		T
S				E				P		A			A		U				C		S
			20 B		21 I	M	P	L	E	M	E	N	T		22 F	I	L	T	E	R	
	23 B		R		N			O					I		F				S		24 W
25 D	I	R	E	C	T	O	R	Y			26 S	O	C	K	E	T		27 U	S	E	R
	N		A		E						T				R		28 P				A
29 P	A	C	K	A	G	E			30 C	L	A	S	31 S			32 L	O	O	P		P
	R				E				L		T		L				R				P
33 B	Y	34 T	E		35 R	E	36 S	T	O	R	E		E		37 E	38 X	T	R	39 E	M	E
		A		40 I			E		S				E			M			X		R
41 M	A	N	I	F	E	S	T		42 E	N	C	A	P	S	U	L	A	T	E		S

18 Távoli telepítés az RMI segítségével

Elosztott számítások



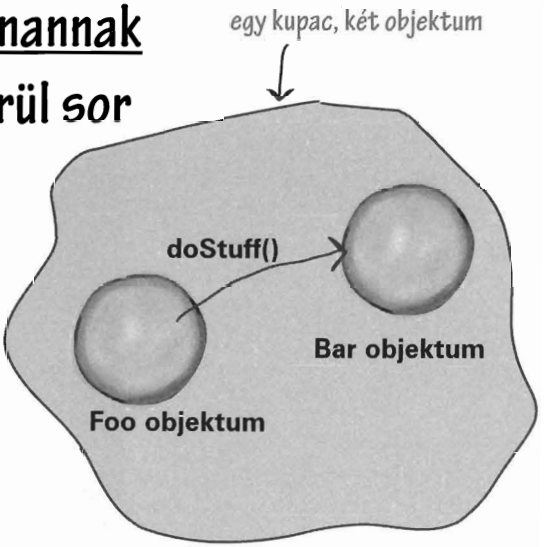
Távol lenni nem feltétlenül rossz. Természetesen *könnyebb*, ha az alkalmazásod minden része egy helyen, egy kupacban, egy Java virtuális gép uralma alatt él, de ez nem mindig lehetséges – vagy éppen kívánatos. Mi van, ha az alkalmazás bonyolult számításokat végez, de a végfelhasználóknak csak egy szerény kis Java-képes eszközük van? Mi van, ha az alkalmazásnak egy adatbázisból kell adatokat kiolvasnia, de biztonsági megfontolások miatt csak a kiszolgálón található kód férhet hozzá az adatbázishoz? El tudsz képzelni mondjuk egy nagy e-kereskedelmi háttérprogramot, amelynek egy tranzakció-kezelő rendszeren belül kell futnia? Néha előfordul, hogy az alkalmazásod egy részének *muszáj* egy kiszolgálón működnie, míg a másik része (általában egy ügyfél) egy *másik* számítógépen kell fusson. Ebben a fejezetben a Java elképesztően egyszerű RMI (Remote Method Invocation, távoli tagfüggvényhívás) technológiájával ismerkedhetsz meg. Emellett pillantást vetünk a kiszolgálói kisalkalmazásokra (servlet-ekre), a vállalati szintű Java-babszemekre (Enterprise Java Beans, EJB), a Jini-re, valamint arra is, hogy az EJB és a Jini hogyan *támaszkodik* az RMI-re. A könyvet a Javában elkészíthető egyik legklasszabb dolog megírásával zárjuk: egy *univerzális szolgáltatásböngészőt* fogunk készíteni.



A tagfüggvényhívásokra mindig ugyanannak a kupacnak két objektuma között kerül sor

A könyvben eddig minden tagfüggvényt egy olyan objektumra hívtunk meg, ami ugyanazon a virtuális gépen futott, mint a hívó. Más szavakkal, a hívó objektum és a címzett (az objektum, amelyre a tagfüggvényt meghívtuk) ugyanabban a kupacban élt.

```
class Foo {
    void go() {
        Bar b = new Bar();
        b.doStuff();
    }
    public static void main (String[] args) {
       Foo f = new Foo();
       f.go();
    }
}
```

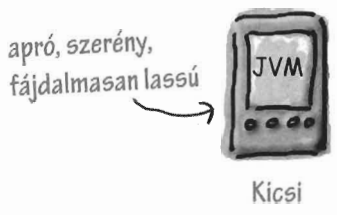


A legtöbb alkalmazásban, amikor egy objektum meghív egy tagfüggvényt egy másik objektumra, mindkét objektum ugyanabban a kupacban található. Más szavakkal, mindkettő ugyanazon a JVM-en belül fut.

A fenti kódban tudjuk, hogy az *f* által hivatkozott Foo-példány és a *b* által hivatkozott Bar objektum ugyanabban a kupacban található, és ugyanaz a JVM futtatja őket. Ne feledd, hogy a JVM dolga, hogy belegyömöszölje azokat a biteket a hivatkozó változókba, amelyek azt jelképezik, hogy *miként lehet elérni egy objektumot a kupacban*. A JVM mindig tudja, hogy hol vannak az egyes objektumok, és hogyan férhet hozzájuk – de csak a *saját* kupacában található hivatkozásokat ismeri! Egy adott számítógépen működő JVM például nem tudhat egy *másik* gépen futó JVM kupacterületéről, sőt még az *ugyanazon* a gépen futó más JVM-ekről sem tudhat semmit. Nem számít, hogy két JVM fizikailag ugyanazon a gépen vagy eltérő gépeken található, csak az, hogy a két JVM különbözik egymástól.

Mi van, ha egy másik gépen futó objektumra szeretnék meghívni egy tagfüggvényt?

Azt már tudod, hogyan szerezhetsz információt egy másik számítógépről – csatolók és bemeneti-kimeneti adatfolyamok segítségével. Megnyitsz egy csatolókapcsolatot a másik géppel, létrehozol egy OutputStream-et, és adatokat írsz bele. De mi van, ha egy *tagfüggvényt szeretnél meghívni* egy másik gépen, egy másik Java virtuális gépen futó dologra? Természetesen mindig van rá lehetőség, hogy írj egy saját protokollt, így amikor adatokat küldesz egy kiszolgálói csatolóra (ServerSocket), a kiszolgáló fel tudja dolgozni azokat, ki tudja találni, hogy mit akarsz, elvégezheti a munkát, és visszaküldheti az eredményt egy másik adatfolyamban. Ez azonban elég fáradságos. Gondolj bele, mennyivel szebb lenne, ha csak egy hivatkozást kellene szerezned a másik gépen levő objektumra, és máris meghívhatnál egy tagfüggvényt.





Képzelj el két számítógépet...

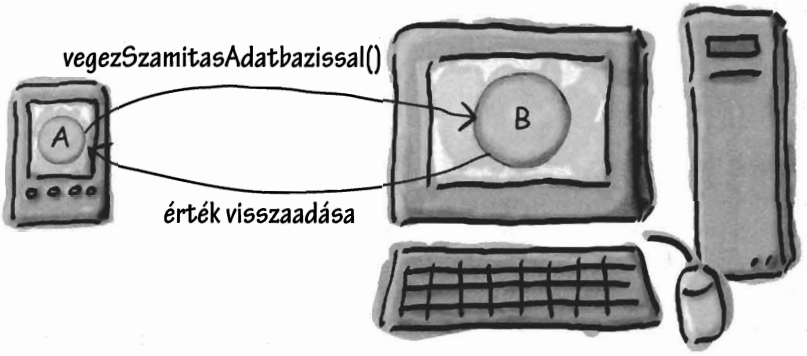
A Nagy gépen van valami, amire a Kicsinek szüksége van: számítási teljesítmény.
A Kicsi adatokat szeretne küldeni a Nagynak, hogy a Nagy végezhesse el a bonyolultabb számításokat.
A Kicsi azt szeretné, hogy csak egy tagfüggvényt kelljen meghívnia...

```
double vegezSzamitasAdatbazissal(KiszamitSzamok szamok)
```

...és megkapja az eredményt.
De hogyan juthat hozzá a Kicsi egy hivatkozáshoz, ami a Nagyon levő objektumra mutat?

A Kicsin futó A objektum a Nagyon futó B objektumra szeretne meghívni egy tagfüggvényt

A kérdés az, hogy miként vehetsz rá egy objektumot, hogy egy másik számítógépen levő (vagyis egy másik kupacban levő/másik JVM-en futó) objektumra hívjon meg egy tagfüggvényt?



Csakhogy ezt nem teheted meg.

Pontosabban közvetlenül nem. Egy másik kupacban levő dologra nem szerezhetsz hivatkozást. Ha ezt mondod...

Dog d = ???

...akkor bármire is hivatkozzon a *d*, annak ugyanazon a kupacterületen kell lennie, mint az utasítást kiadó kódnak.

Tegyük fel azonban, hogy valami olyasmit szeretnél tervezni, ami csatolókat és bemeneti-kimeneti adatfolyamokat használ a szándékaid közlésére (vagyis hogy egy tagfüggvényt szeretnél meghívni egy másik számítógépen futó objektumra), mégis *olyan*, mintha helyi tagfüggvényhívást hajtanál végre.

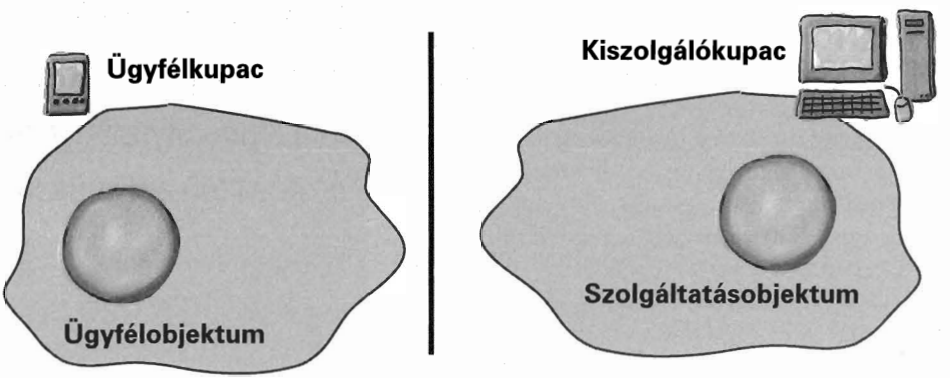
Más szavakkal, egy *távoli* objektumra (vagyis egy máshol található kupacban levő objektumra) szeretnél meghívni egy tagfüggvényt, de olyan kóddal, ami *úgy tesz*, mintha a tagfüggvényt egy helyi objektumra hívná meg. A jól bevált, szokványos tagfüggvényhívás egyszerűségét szeretnéd, de a távoli tagfüggvényhívás hatékonyságát. Ez a cél – és ezt biztosítja számodra az RMI (Remote Method Invocation, távoli tagfüggvényhívás)!

De ne siessünk előre: először gondold végig, te hogyan terveznéd meg az RMI-t, ha magadra lennél utalva. Ha megérted, mire lenne szükség, könnyebben felfoghatod az RMI működését is.

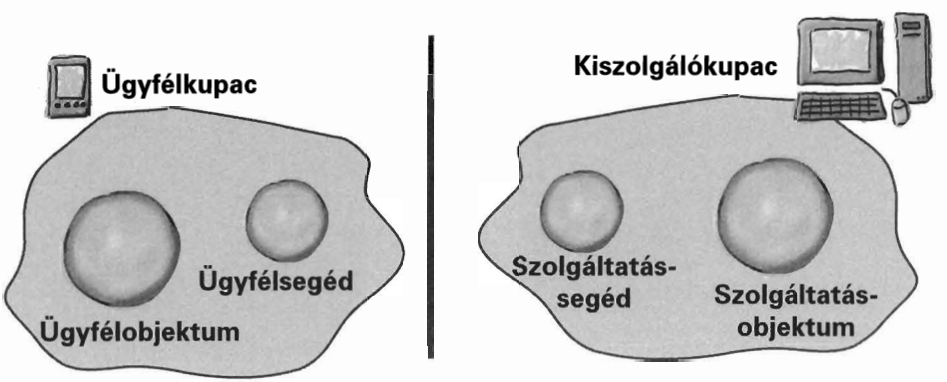
Terv távoli tagfüggvényhívásokhoz

Négy dolgot kell létrehozni: egy kiszolgálót, egy ügyfelet, és kiszolgálósegédet és egy ügyfélsegédet.

1. Létrehozni az ügyfél- és kiszolgálóalkalmazásokat. A kiszolgálóalkalmazás az a **távoli szolgáltatás**, ahol az az objektum található, amelyiknek a tagfüggvényét az ügyfél meg szeretné hívni.



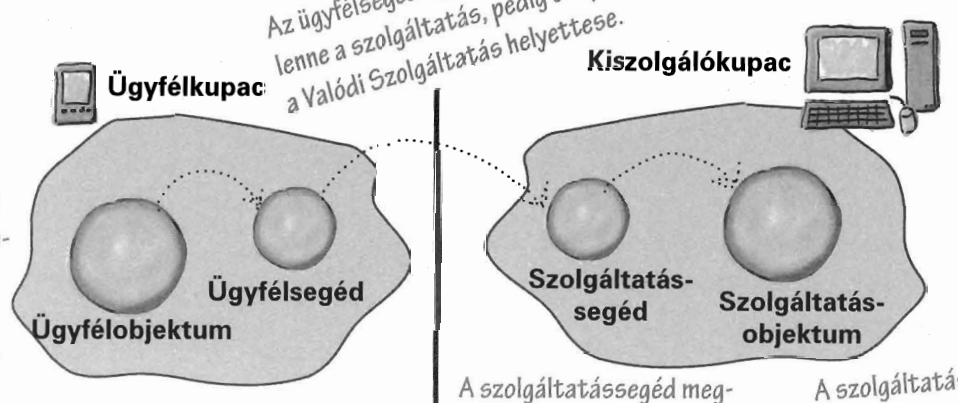
2. Létrehozni az ügyfél- és kiszolgálósegédeket. A „segédek" kezelik az alacsonyszintű hálózati és I/O-műveleteket, hogy az ügyfél és a szolgáltatás úgy tehessen, mintha ugyanabban a kupacban lennének.



A „segédek" szerepe

A „segédek" (helper) azok az objektumok, amelyek a tényleges kommunikációt bonyolítják. Ők teszik lehetővé az ügyfélnek, hogy *úgy tehessen*, mintha egy helyi objektum tagfüggvényét hívná meg. Valójában *azt is teszi*: az ügyfél az ügyfélsegéd egy tagfüggvényét hívja meg, *mintha az ügyfélsegéd lenne a tényleges szolgáltatás.* Az *ügyfélsegéd tehát a Valódi Szolgáltatás helyettese (proxy).*
Más szavakkal, az ügyfélobjektum azt *hiszi*, hogy a távoli szolgáltatás egy tagfüggvényét hívja, mert az ügyfélsegéd *úgy tesz*, mintha ő lenne a szolgáltatásobjektum, ***vagyis az a dolog, amelynek a tagfüggvényét az ügyfél meg akarja hívni***.
Az ügyfélsegéd azonban valójában nem a távoli szolgáltatás. Bár *úgy tesz*, mintha az lenne (mivel van egy ugyanolyan tagfüggvénye, mint az általa helyettesített szolgáltatásnak), ténylegesen semmivel sem rendelkezik az ügyfél által várt tagfüggvénylogikából. Az ügyfélsegéd ehelyett kapcsolatba lép a kiszolgálóval, átadja neki a tagfüggvényhívással kapcsolatos információkat (a tagfüggvény nevét, az argumentumokat stb.), és várja a kiszolgáló válaszát.
A kiszolgáló oldalán a szolgáltatássegéd fogadja az ügyfélsegédtől érkező kérelmet (egy csatolókapcsolaton keresztül), kicsomagolja a hívással kapcsolatos információkat, majd meghívja a *valódi* szolgáltatásobjektum *valódi* tagfüggvényét. A szolgáltatásobjektum számára tehát a hívás helyi, hiszen a szolgáltatássegédtől érkezik, nem egy távoli ügyféltől. A szolgáltatássegéd megkapja a visszatérési értéket a szolgáltatástól, becsomagolja, és visszaküldi (egy csatoló kimeneti adatfolyamán keresztül) az ügyfélsegédnek. Az ügyfélsegéd kicsomagolja az információt, és átadja a visszatérési értéket az ügyfélobjektumnak.

Az ügyfélobjektum úgy tesz, mintha távoli tagfüggvényhívásokat bonyolítana, pedig valójában a helyi kupacban található „helyettes" objektum tagfüggvényeit hívja, és ez az objektum gondoskodik minden alacsonyszintű csatoló- és adatfolyam-műveletről.



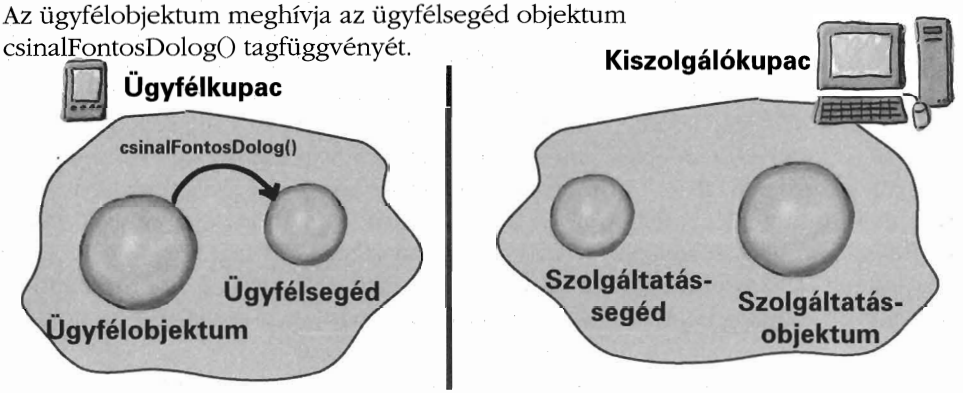
Az ügyfélobjektum azt hiszi, hogy a Valódi Szolgáltatással beszél. Azt hiszi, hogy az ügyfélsegéd valóban képes végrehajtani a kért feladatot.

A szolgáltatássegéd megkapja a kérelmet az ügyfélsegédtől, kicsomagolja, és meghívja a Valódi Szolgáltatás tagfüggvényét.

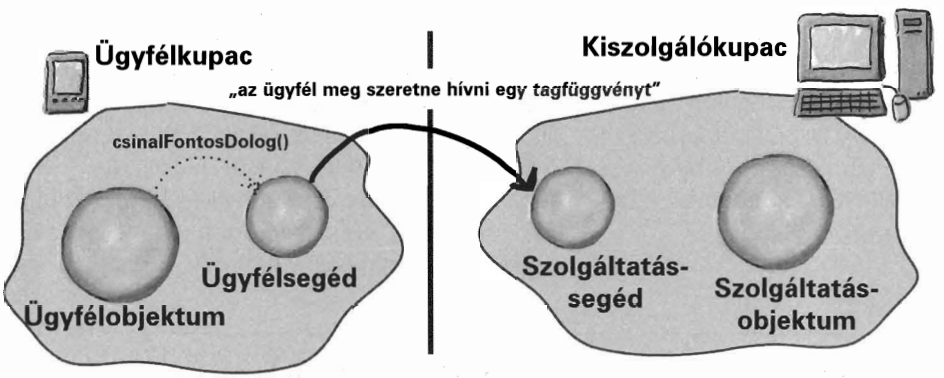
A szolgáltatásobjektum a Valódi Szolgáltatás: az az objektum, amelyiknek a tagfüggvénye a tényleges munkát végzi.

A tagfüggvényhívás működése

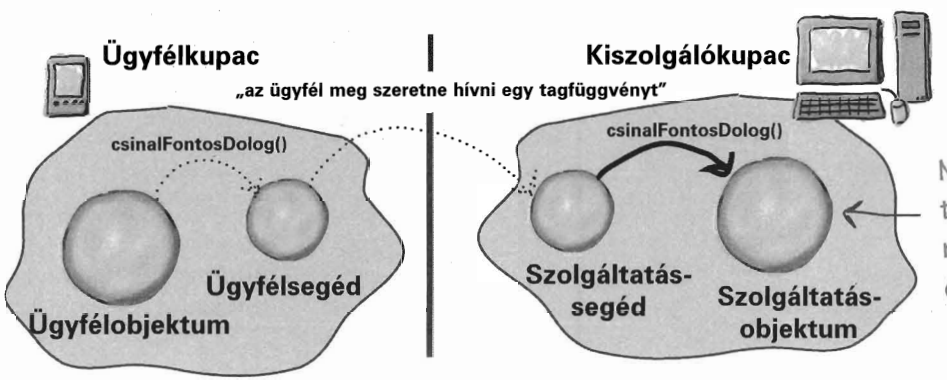
1. Az ügyfélobjektum meghívja az ügyfélsegéd objektum csinalFontosDolog() tagfüggvényét.



2. Az ügyfélsegéd becsomagolja a hívással kapcsolatos információkat (argumentumok, a tagfüggvény neve stb.), és átküldi a hálózaton a szolgáltatássegédnek.



3. A szolgáltatássegéd kicsomagolja az ügyfélsegédtől kapott információkat, kideríti, melyik tagfüggvényt kell meghívni (és melyik objektumra), majd meghívja a valódi tagfüggvényt a valódi szolgáltatásobjektumra.



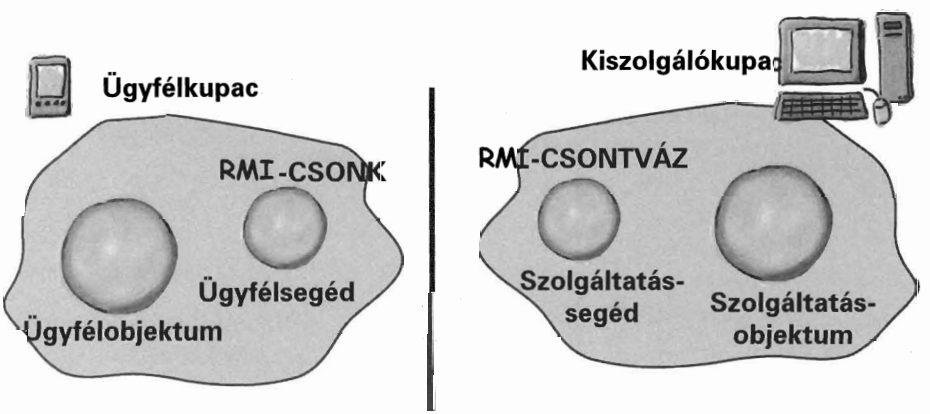
Ne feledd, hogy ez a VALÓDI tagfüggvénylogikát tartalmazó tagfüggvény – tehát ő végzi a tényleges munkát!

A Java RMI biztosítja az ügyfél- és szolgáltatássegéd objektumokat

A Javában az RMI felépíti helyetted az ügyfélsegéd és szolgáltatássegéd (kiszolgálósegéd) objektumokat, sőt azt is tudja, hogyan érheti el, hogy az ügyfélsegéd úgy nézzen ki, mint a Valódi Szolgáltatás. Más szavakkal, az RMI tudja, hogyan láthatja el az ügyfélsegéd objektumot ugyanazokkal a tagfüggvényekkel, mint amelyeket meg szeretnél hívatni a távoli szolgáltatással.
Ezenkívül az RMI teljes futásidejű rendszert biztosít ahhoz, hogy ez működjön, beleértve egy keresőszolgáltatást is, amelynek a segítségével az ügyfél megtalálhatja az ügyfélsegédet (a Valódi Szolgáltatás helyettesét).
Az RMI-nek köszönhetően neked *egyáltalán* nem kell hálózat- vagy I/O-kezelő kódot írnod. Az ügyfél pontosan úgy hívhatja meg a távoli (vagyis a Valódi Szolgáltatás birtokában levő) tagfüggvényeket, mint a saját JVM-jén futó objektumok tagfüggvényeit.
Illetve majdnem pontosan úgy.
Egy különbség azért van az RMI-hívások és a helyi (normál) tagfüggvényhívások között. Ne feledd, hogy bár az ügyfél számára a tagfüggvényhívás helyinek tűnik, az ügyfélsegédnek attól még át kell küldenie a hívást a hálózaton keresztül. Tehát hálózati és bemeneti-kimeneti műveletekre van szükség. És mit tudsz a hálózati és bemeneti-kimeneti műveletekről?
Azt, hogy kockázatosak!
Állandóan kivételeket váltanak ki.

Az ügyfélnek tehát tisztában kell lennie a kockázattal. Tudomásul kell vennie, hogy amikor egy távoli tagfüggvényt hív meg, akkor *végső soron* csatolókkal és adatfolyamokkal dolgozik, annak ellenére, hogy számára csupán egy helyetteshez vagy segédobjektumhoz intézett helyi hívásról van szó. Az ügyfél eredeti hívása *helyi*, de a helyettes ezt *távoli* hívássá alakítja. A távoli hívás csupán annyit jelent, hogy egy másik JVM-en futó objektumra hívsz meg egy tagfüggvényt. Az, hogy a hívással kapcsolatos információk *hogyan* kerülnek át az egyik JVM-ről a másikra, a segédobjektumok által használt protokolltól függ.
Az RMI-ben választhatsz, hogy a JRMP vagy az IIOP protokollt szeretnéd használni. A JRMP az RMI „belső" („natív") protokollja, amelyet kifejezetten Java–Java távoli hívásokhoz fejlesztettek ki. Az IIOP ezzel szemben a CORBA (Common Object Request Broker Architecture) protokollja, és olyan dolgokhoz is intézhetsz vele távoli hívásokat, amelyek nem feltétlenül Java-objektumok. A CORBA használata általában *sokkal* nehézkesebb, mint az RMI-é, mert ha nem Java működik a kapcsolat mindkét végén, akkor rengeteg fordítási és átalakítási műveletet kell elvégezni.
Minket szerencsére csak a Java és Java közötti műveletek érdekelnek, ezért maradunk a figyelemre méltóan egyszerűen használható RMI-nél.

Az RMI-ben az ügyfélsegéd egy „csonk" (stub), a kiszolgálósegéd pedig egy „csontváz" (skeleton).



A Távoli Szolgáltatás elkészítése

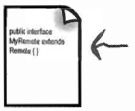
Az alábbiakban egy **áttekintés** következik a (kiszolgálón futó) távoli szolgáltatás elkészítéséhez szükséges öt lépésről. Ne aggódj, minden lépést részletesen elmagyarázunk a következő oldalakon.



Kiszolgáló

Első lépés:
Távoli felület készítése

A távoli felület határozza meg azokat a tagfüggvényeket, amelyeket egy ügyfél távolról meghívhat. Ezt a felületet fogja az ügyfél a szolgáltatás többalakú osztálytípusaként használni. Mind a csonknak, mind a tényleges szolgáltatásnak meg kell valósítania ezt a felületet!



MyRemote.java

Ez a felület határozza meg azokat a távoli tagfüggvényeket, amelyeket az ügyfelekkel meg szeretnél hívatni.

Második lépés:
Távoli megvalósítás készítése

Ez az osztály végzi a Valódi Munkát. Ebben az osztályban található a távoli felület által meghatározott távoli tagfüggvények tényleges megvalósítása. Ez az az objektum, amelynek a tagfüggvényeit az ügyfél meg szeretné hívni.



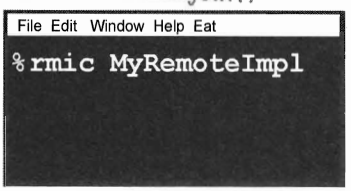
MyRemoteImpl.java

A Valódi Szolgáltatás. Ez az osztály tartalmazza azokat a tagfüggvényeket, amelyek a tényleges munkát végzik, és ez valósítja meg a távoli felületet.

Harmadik lépés:
A **csonkok** és **csontvázak** előállítása az rmic segítségével

Ezek az ügyfél és a kiszolgáló „segédjei". Ezeket az osztályokat nem neked kell megírnod, és az őket előállító forráskódba sem kell soha belepillantanod. A Java-fejlesztőcsomagban megtalálható rmic segédprogramot kell elindítanod, és az mindenről automatikusan gondoskodik.

Az rmic futtatása a tényleges szolgáltatást megvalósító osztályon...



...két új osztályt köp ki a segédobjektumok számára



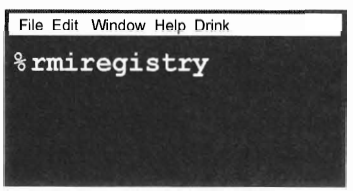
MyRemoteImpl_Stub.class



MyRemoteImpl_Skel.class

Negyedik lépés:
Az **RMI-jegyzék** (rmiregistry) elindítása

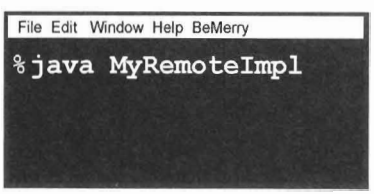
Az RMI-jegyzék olyan, mint a telefonkönyv oldalai. A felhasználó innen szerzi majd meg a helyettest (az ügyfélcsonkot vagy segédobjektumot).



Ezt egy önálló terminálban futtasd

Ötödik lépés:
A **távoli szolgáltatás** elindítása

A szolgáltatásobjektumot üzembe kell helyezned. A szolgáltatást megvalósító osztály létrehozza a szolgáltatás egy példányát, és bejegyzi az RMI-jegyzékbe. A bejegyzés elérhetővé teszi a szolgáltatást az ügyfelek számára.



Első lépés: Távoli felület készítése



MyRemote.java

① Bővítjük a java.rmi.Remote felületet

A Remote (Távoli) egy „jelzőfelület", ami azt jelenti, hogy nincsenek tagfüggvényei. Az RMI számára azonban különleges jelentéssel bír, ezért mindenképpen követned kell ezt a szabályt. Vedd észre, hogy „bővítésről" (extends) beszélünk: egy felület számára megengedett egy másik felület *bővítése*.

```
public interface MyRemote extends Remote {
```

A felületednek be kell jelentenie, hogy távoli tagfüggvényhívásokra szolgál. Egy felület nem valósíthat meg semmit, de bővíthet más felületeket.

② Bejelentjük, hogy minden tagfüggvény RemoteException-t vált ki

Az ügyfél a távoli felületet használja a szolgáltatás többalakú típusaként. Más szavakkal, az ügyfél valami olyasmire hív meg tagfüggvényeket, ami megvalósítja a távoli felületet. Ez a valami természetesen a csonk, és mivel a csonk végzi a hálózati és a bemeneti-kimeneti műveleteket, mindenféle Rossz Dolog történhet. Az ügyfélnek tudomásul kell vennie a kockázatokat, azzal, hogy kezeli vagy bevezeti a távoli kivételeket. Ha egy felület tagfüggvényei kivételeket vezetnek be, az ebbe a típusba (a felület típusába) tartozó hivatkozásokra tagfüggvényeket meghívó bármely kódnak ugyancsak kezelnie kell vagy be kell vezetnie a kivételeket.

```
import java.rmi.*;
```

A Remote felület a java.rmi-ben található.

```
public interface MyRemote extends Remote {
    public String sayHello() throws RemoteException;
}
```

Minden távoli tagfüggvényhívás „kockázatosnak" minősül. A RemoteException bevezetése minden tagfüggvényre arra kényszeríti az ügyfelet, hogy odafigyeljen, és tudomásul vegye, hogy a dolog esetleg nem fog működni.

③ Gondoskodunk róla, hogy az argumentumok és visszatérési értékek primitívek vagy sorosíthatók (Serializable) legyenek

Egy távoli tagfüggvény argumentumainak és visszatérési értékeinek vagy primitíveknek, vagy sorosíthatóknak (Serializable) kell lenniük. Gondold végig: a távoli tagfüggvény minden argumentumát be kell csomagolni és át kell vinni a hálózaton, és ez sorosítás segítségével történik. Ugyanez a helyzet a visszatérési értékekkel is. A primitíveket, karakterláncokat és az API-ban található típusok többségét (beleértve a tömböket és a gyűjteményeket is) nyugodtan használhatod. Ha saját típusokat adsz át, ügyelj rá, hogy az osztályaid megvalósítsák a Serializable felületet.

```
public String sayHello() throws RemoteException;
```

Ezt a visszatérési értéket a hálózati kábelen át küldi vissza a kiszolgáló az ügyfélnek, ezért sorosíthatónak kell lennie. Így történik az argumentumok és visszatérési értékek becsomagolása és elküldése.

Második lépés: Távoli megvalósítás készítése



MyRemoteImpl.java

1 Megvalósítjuk a Remote felületet

A szolgáltatásodnak meg kell valósítania a távoli felületet – azt, amelyiknek a tagfüggvényeit az ügyfél majd meg fogja hívni.

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
    public String sayHello() {
        return "Server says, 'Hey'";
    }
    // további kód az osztályban
}
```

A fordító ügyel rá, hogy a felület minden tagfüggvényét megvalósítsd. Ebben az esetben csak egy tagfüggvényünk van.

2 Bővítjük az UnicastRemoteObject osztályt

Ahhoz, hogy az objektumod távoli szolgáltatásobjektumként működni tudjon, a „távolisággal" kapcsolatos képességekkel kell rendelkeznie. A legegyszerűbb megoldás, ha az UnicastRemoteObject osztályt bővíted (a java.rmi.server csomagból), majd erre az osztályra (vagyis az ősosztályra) bízod a munka elvégzését.

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
```

3 Írunk egy argumentum nélküli konstruktort, amely bevezet egy RemoteException-t

Az új ősosztályoddal, az UnicastRemoteObject-tel van egy aprócska gond: a konstruktora RemoteException kivételt vált ki. Ezt csak egyetlen módon kezelheted: be kell vezetned egy konstruktort a távoli megvalósításod számára, hogy legyen egy hely, ahol bejelentheted a RemoteException-t. Ne feledd, hogy amikor példányosítasz egy osztályt, az alaposztálya (ősosztálya) konstruktora mindig meghívódik. Ha az alaposztályod konstruktora kivételt dob, nincs más választásod, mint hogy a te konstruktorodban is bejelentsd a kivétel dobását.

```
public MyRemoteImpl() throws RemoteException { }
```

A konstruktorba semmit nem kell tenned, csak valahogy be kell jelentened, hogy az ősosztályod konstruktora kivételt vált ki.

4 Bejegyezzük a szolgáltatást az RMI-jegyzékbe

Most, hogy van egy távoli szolgáltatásod, hozzáférhetővé kell tenned a távoli ügyfelek számára. Ezt úgy teheted meg, hogy példányosítod a szolgáltatást, és felveszed az RMI-jegyzékbe (aminek futnia kell, különben ez a kódsor kudarcot vall). Amikor bejegyezteted a megvalósító objektumot, az RMI rendszer valójában a *csonkot* teszi a jegyzékbe, mivel ez az, amire az ügyfélnek valójában szüksége van. A szolgáltatást a java.rmi.Naming osztály statikus rebind() tagfüggvényével jegyeztetheted be.

```
try {
    MyRemote service = new MyRemoteImpl();
    Naming.rebind("Remote Hello", service);
} catch(Exception ex) {...}
```

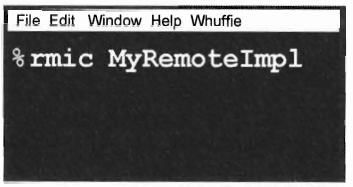
Adj a szolgáltatásnak egy nevet és vedd fel az RMI-jegyzékbe. Amikor hozzákötöd a szolgáltatásobjektumot, az RMI a szolgáltatás helyére behelyettesíti a csonkot, és azt teszi a jegyzékbe.

Harmadik lépés: A csonkok és csontvázak előállítása

1 Futtatjuk az rmic-et a távoli megvalósító osztályra (nem a távoli felületre)

Az rmic segédprogram, ami a Java szoftverfejlesztő készlet része, a szolgáltatás megvalósításából két új osztályt hoz létre: a csonkot és a csontvázat. Az osztályok elnevezésekor a távoli megvalósítás nevét veszi alapul, és a végéhez hozzábiggyeszti a _Stub, illetve a _Skeleton tagot. Az rmic más választási lehetőségeket is kínál: megteheted, hogy nem állíttatsz elő csontvázat; megnézheted, hogy miként fest az említet osztályok forráskódja; sőt akár az IIOP-t is használhatod protokollként. Itt azt a módszert követjük, amit általában te is fogsz. Az osztályok az aktuális könyvtárban (tehát ahová a cd paranccsal éppen váltottál) kötnek ki. Ne feledd, hogy az *rmic*-nek látni kell a megvalósító osztályodat, ezért az rmic-t valószínűleg a távoli megvalósítást tartalmazó könyvtárból fogod futtatni. (Itt az egyszerűség kedvéért szándékosan nem használunk csomagokat, de a valóságban figyelembe kell venned a csomagkönyvtár-szerkezetet és a teljesen minősített neveket is.)

Figyeld meg, hogy a végére nem írod oda, hogy „.class", csak az osztálynevet használod.



A segédprogram két új osztályt köp ki a segédobjektumok számára.



MyRemoteImpl_Stub.class

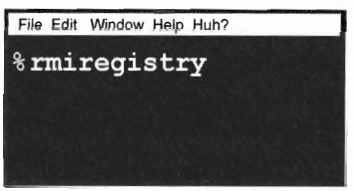


MyRemoteImpl_Skel.class

Negyedik lépés: az rmiregistry futtatása

1 Megnyitunk egy terminált, és elindítjuk az rmiregistry segédprogramot.

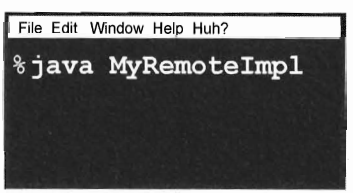
Ügyelj rá, hogy a segédprogramot egy olyan könyvtárból indítsd el, amely hozzáfér az osztályaidhoz. A legegyszerűbb, ha a „classes" könyvtárból indítod el.



Ötödik lépés: A szolgáltatás elindítása

1 Nyiss egy újabb terminált, és indítsd el a szolgáltatást.

Ezt megteheted a távoli megvalósító osztály main() tagfüggvényéből, de egy önálló indítóosztályból is. Egyszerű példánkban az indítókódot a megvalósító osztályba tettük, egy olyan main tagfüggvénybe, amely példányosítja az objektumot, és bejegyezteti azt az RMI-jegyzékbe.



A kiszolgálóoldal teljes kódja

Kiszolgáló



A távoli felület:

A RemoteException és a Remote felület a java.rmi csomagban található.

```
import java.rmi.*;

public interface MyRemote extends Remote {

    public String sayHello() throws RemoteException;
}
```

A felületednek a java.rmi-Remote-ot KELL bővítenie.

Minden távoli tagfüggvényednek be kell vezetnie egy RemoteException-t.

A távoli szolgáltatás (a megvalósítás):

Az UnicastRemoteObject a java.rmi.server csomagban található.

Az UnicastRemoteObject bővítése a legegyszerűbb módja egy távoli objektum létrehozásának.

```
import java.rmi.*;
import java.rmi.server.*;

public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {

    public String sayHello() {
        return "Server says, 'Hey'";
    }

    public MyRemoteImpl() throws RemoteException { }

    public static void main (String[] args) {

        try {
            MyRemote service = new MyRemoteImpl();
            Naming.rebind("Remote Hello", service);
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
```

Természetesen a felület összes tagfüggvényét meg kell valósítanod, de vedd észre, hogy NEM kell bevezetned a RemoteException-t.

Meg KELL valósítanod a távoli felületet!!!

Az ősosztályod konstruktora (az UnicastRemoteObject-é) bevezet egy kivételt, ezért NEKED is írnod kell egy konstruktort, mert a kivétel azt jelenti, hogy a konstruktorod kockázatos kódot hív meg (az őskonstruktorát).

Létrehozzuk a távoli objektumot, majd „hozzákötjük" (bind) az rmiregistryhez a statikus Naming.rebind() tagfüggvénnyel. Az objektumot az ügyfeleknek a bejegyzett név alapján kell kikeresniük az RMI-jegyzékből.

Hogyan jut hozzá az ügyfél a csonk objektumhoz?

Az ügyfélnek meg kell szereznie a csonk objektumot, mivel ez az, aminek a tagfüggvényeit meg fogja hívni. Ekkor kap szerepet az RMI-jegyzék. Az ügyfél „keresést" (lookup) végez, mintha fellapozna egy telefonkönyvet, és lényegében azt mondja, hogy „itt egy név, és meg szeretném találni a hozzá tartozó csonkot".

A lookup() a Naming osztály statikus tagfüggvénye.

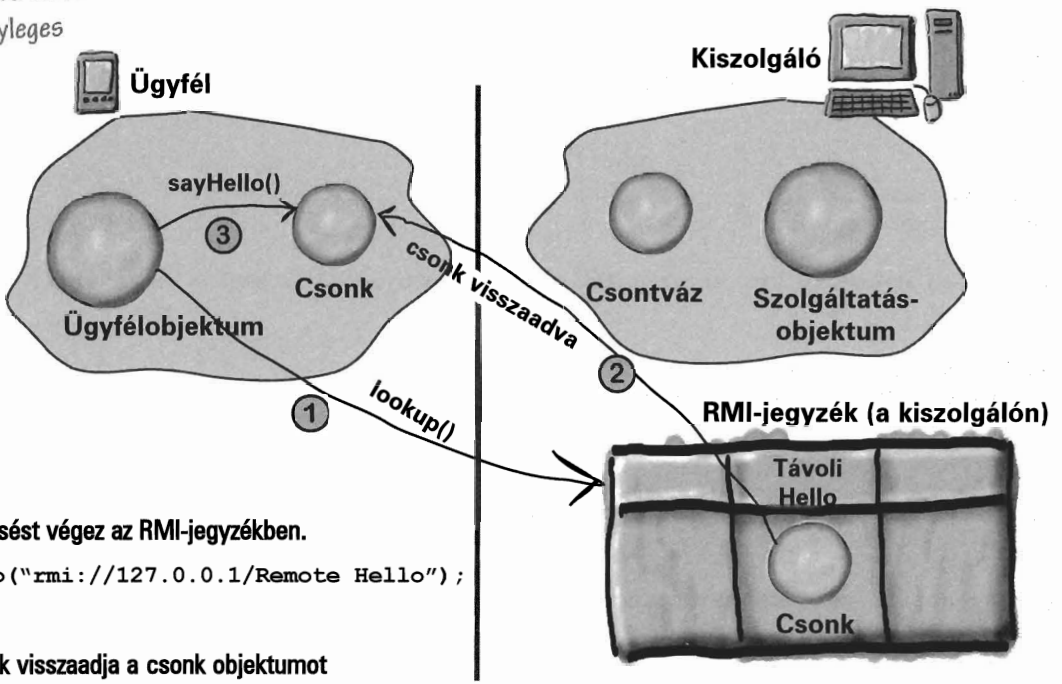
Itt annak a névnek kell állnia, amelyen a szolgáltatás be lett jegyezve.

```
MyRemote service = (MyRemote)  Naming.lookup("rmi://127.0.0.1/Remote Hello");
```

Az ügyfél a szolgáltatás típusaként mindig a távoli megvalósítást használja. Valójában az ügyfélnek soha nem kell tudnia a távoli szolgáltatás tényleges osztálynevét.

Át kell alakítani a felület típusára, mivel a keresőfüggvény Object típusú értéket ad vissza.

Ide jön a gépneved vagy az IP-címed.



1 Az ügyfél keresést végez az RMI-jegyzékben.

```
Naming.lookup("rmi://127.0.0.1/Remote Hello");
```

2 Az RMI-jegyzék visszaadja a csonk objektumot
(a keresőfüggvény visszatérési értékeként), az RMI pedig automatikusan visszafejti a sorosított csonkot. A csonk osztállyal (amelyet az rmic állított elő a számodra) rendelkezned KELL az ügyfélen, különben a csonk visszafejtése nem sikerül.

3 Az ügyfél meghív egy tagfüggvényt a csonkra, mintha a csonk lenne a valódi szolgáltatás.

Hogyan jut hozzá az ügyfél a csonk osztályhoz?

Most jön az érdekes kérdés. Az ügyfélnek valamilyen módon hozzá kell jutnia a csonk osztályhoz (amelyet korábban az rmic segítségével állítottál elő), amikor a keresést végzi, klülönben a csonk nem fejthető vissza a sorosításból az ügyfélen, és az egész művelet kudarcot vall.
Egy egyszerű rendszeren megteheted, hogy saját kezűleg kézbesíted a csonk osztályt az ügyfélnek. Van persze egy sokkal klasszabb módszer is – dinamikus osztályletöltésnek hívják –, de ennek tárgyalása túlmutat könyvünk keretein. Ha érdekel a dolog, röviden arról van szó, hogy a csonk objektum (vagy bármely más sorosított objektum) egy „URL-bélyeget" kap, ami elárulja az ügyfélen található RMI rendszernek, hogy hol találja az említett objektum osztályfájlját. Az objektum visszafejtése során aztán, ha az RMI helyben nem találja az osztályt, ennek az URL-nek a segítségével végrehajt egy Get HTTP-műveletet, hogy megszerezze az osztályfájlt. Az osztályfájlok rendelkezésre bocsátásához tehát egy egyszerű webkiszolgálóra van szükség, és emellett az ügyfélen is meg kell változtatni pár biztonsági beállítást. A dinamikus osztályletöltésnek persze egyéb trükkjei is vannak, de a lényeg dióhéjban ez.

Az ügyfél teljes kódja

A Naming osztály a java.rmi csomagban található (és az RMI-jegyzékben való kereséshez szükséges).

```
import java.rmi.*;

public class MyRemoteClient {
   public static void main (String[] args) {
        new MyRemoteClient().go();
   }

  public void go() {

     try {
        MyRemote service = (MyRemote) Naming.lookup("rmi://127.0.0.1/RemoteHello");

        String s = service.sayHello();

        System.out.println(s);
      } catch(Exception ex) {
        ex.printStackTrace();
      }
   }
}
```

A jegyzékből Object típusú értéket kapsz vissza, ezért ne feledkezz meg az átalakításról.

Itt az IP-cím vagy a gépnév megadása szükséges...

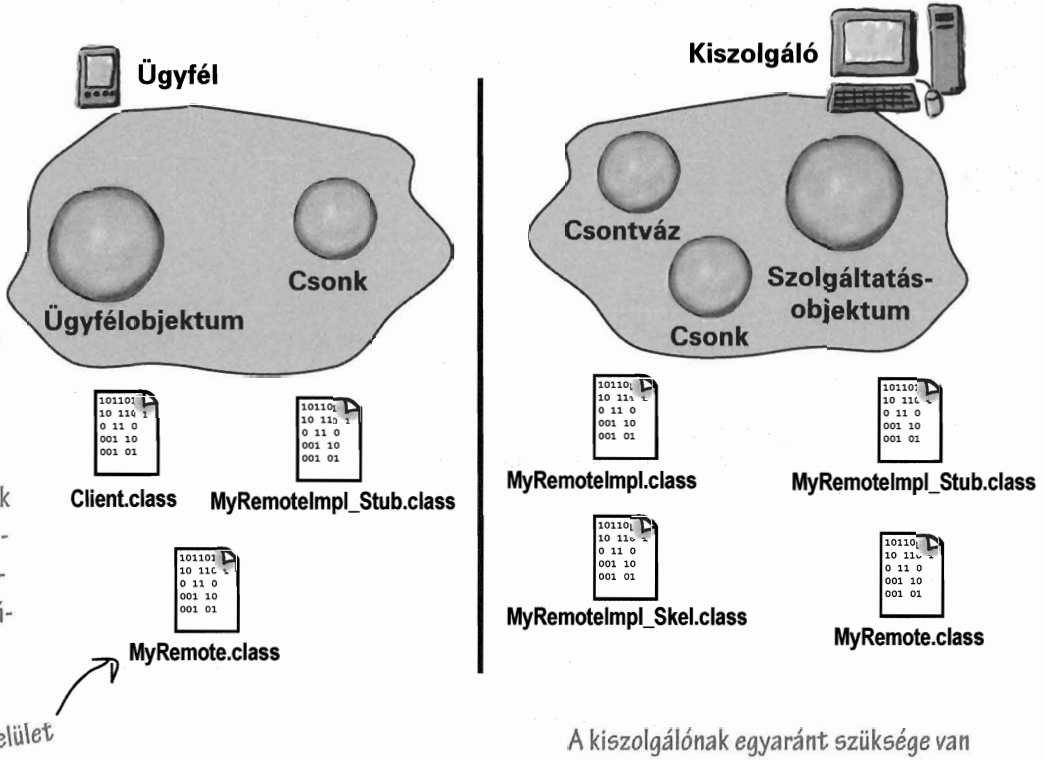
...valamint a szolgáltatás kötéséhez/újrakötéséhez használt név.

Ez pont olyan, mint egy szokványos tagfüggvényhívás! (Csak éppen tudomásul kell venni a RemoteException-t.)

Ügyelj rá, hogy minden gép rendelkezzen a szükséges osztályfájlokkal

Az RMI-vel kapcsolatban a programozók a következő három hibát követik el a leggyakrabban:
1) Elfelejtik elindítani az rmiregistry-t, mielőtt elindítanák a távoli szolgáltatást. (Amikor bejegyzeted a szolgáltatást a Naming.rebind() segítségével, az rmiregistry-nek futnia kell!)
2) Elfelejtik sorosíthatóvá tenni az argumentumokat és visszatérési típusokat. (Ez a futásig nem tűnik fel, mert a fordító nem észleli.)
3) Elfelejtik átadni a csonk osztályt az ügyfélnek.

Ne feledd, hogy az ügyfél a felületet használja a csonk tagfüggvényeinek meghívására. Az ügyfél JVM-jének szüksége van a csonk osztályra, de az ügyfél soha nem hivatkozik rá a kódban. Az ügyfél mindig a távoli felületet használja, mintha az maga a távoli objektum lenne.



A kiszolgálónak egyaránt szüksége van a csonk és a csontváz osztályra, valamint a szolgáltatásra és a távoli felületre. A csonk osztályal azért kell rendelkeznie, mert az RMI azt helyettesíti be a valódi szolgáltatás helyére, amikor a szolgáltatást felveszi az RMI-jegyzékbe.

Hegyezd ki a ceruzád!

Mi volt előbb?



Nézd meg az alábbi eseményeket, és tedd őket abba sorrendbe, ahogy egy Java RMI-alkalmazásban egymást követik!

1.
2.
3.
4.
5.
6.
7.

A csonk elküldi a tagfüggvényhívást a kiszolgálónak.

Az ügyfél keresést végez az RMI-jegyzékben.

Az RMI-jegyzék elindul.

Az ügyfél megszerzi a csonkot az RMI-jegyzékből.

Az ügyfél meghívja a csonk egy tagfüggvényét.

A távoli szolgáltatás bekerül az RMI-jegyzékbe.

Létrejön a távoli szolgáltatás (távoli megvalósítás) egy példánya.

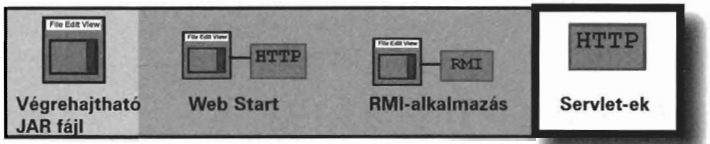


Újratöltve

- Egy adott kupacban levő objektum nem szerezhet normál Java-hivatkozást egy másik kupacban levő (tehát másik JVM-en futó) objektumra.
- A Java Remote Method Invocation (RMI) azt a *hatást* kelti, mintha egy távoli (vagyis másik JVM-en futó) objektum tagfüggvényét hívnád meg, pedig valójában nem.
- Amikor egy ügyfél egy távoli objektum tagfüggvényét hívja, valójában a távoli objektum egy *helyettesének* (proxy) a tagfüggvényét hívja meg. A helyettest „csonk"-nak (stub) hívják.
- A csonk egy ügyfélsegéd-objektum, amely a tagfüggvényhívásokat becsomagolva és a kiszolgálónak elküldve gondoskodik az alacsonyszintű hálózati műveletek (csatolók, adatfolyamok, sorosítás stb.) kezeléséről.
- Ha távoli szolgáltatást szeretnél készíteni (vagyis egy olyan objektumot, amelynek egy távoli ügyfél áttételesen meghívhatja a tagfüggvényeit), először egy távoli felületet kell létrehoznod.
- A távoli felületnek a jav.rmi.Remote felületet kell bővítenie, és minden tagfüggvényének be kell vezetnie egy RemoteException-t.
- A távoli felületet a távoli szolgáltatás valósítja meg.
- A távoli szolgáltatásnak az UnicastRemoteObject osztályt kell bővítenie. (Távoli objektumot más módon is létre lehet hozni, de az UnicastRemoteObject bővítése a legegyszerűbb.)
- A távoli szolgáltatás osztályának mindenképpen rendelkeznie kell egy konstruktorral, és a konstruktornak be kell vezetnie egy RemoteException-t (mivel az alaposztály konstruktora is bevezet egyet).
- A távoli szolgáltatást példányosítani kell, az objektumot pedig be kell jegyeztetni az RMI-jegyzékbe.
- A távoli szolgáltatás bejegyzéséhez a statikus Naming.rebind(„*SzolgaltatasNeve*", *szolgaltatasPeldany*); utasítást használhatod.
- Az RMI-jegyzéknek a távoli szolgáltatással megegyező gépen futnia kell, mielőtt egy távoli objektumot megpróbálhatnál bejegyeztetni a jegyzékbe.
- Az ügyfél a távoli szolgáltatást a statikus Naming.lookup(„rmi://*GepNev*/*SzolgaltatasNev*") hívással keresheti meg.
- Szinte minden, ami az RMI-vel kapcsolatos, kiválthat (a fordító által ellenőrzött) RemoteException-t, beleértve egy szolgáltatás bejegyzését a jegyzékbe, egy szolgáltatás kikeresését onnan, valamint *minden* távoli tagfüggvényhívást, ami az ügyféltől a csonkhoz érkezik.

Jó, de ki használja az RMI-t?





100%-ig helyi **Vegyes** **100%-ig távoli**

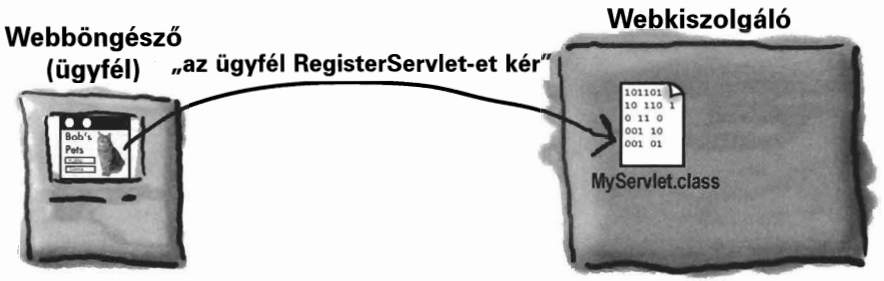
Mi a helyzet a kiszolgálói kisalkalmazásokkal?

A servlet-ek (kiszolgálói kisalkalmazások) olyan Java-programok, amelyek egy HTTP-webkiszolgálón (és annak segítségével) futnak. Amikor egy ügyfél egy webböngésző segítségével kapcsolatba lép egy weboldallal, a webkiszolgáló egy kérelmet kap. Amennyiben a kérelemnek egy Java-servlet segítségére van szüksége, a webkiszolgáló elindítja a servlet kódját (vagy meghívja azt, ha a servlet már fut). A servlet kódja egyszerűen olyan kód, ami a kiszolgálón fut, hogy elvégezze az ügyfél kérelmében igényelt munkát (például információkat mentsen egy szövegfájlba vagy adatbázisba a kiszolgálón). Ha ismered a Perl nyelven írt CGI-parancsfájlokat, akkor pontosan tudod, miről beszélünk. A webfejlesztők mindenre CGI-parancsfájlokat vagy kiszolgálói kisalkalmazásokat használnak, a felhasználók által megadott információk adatbázisba írásától kezdve a webhelyek vitafórumainak működtetéséig.

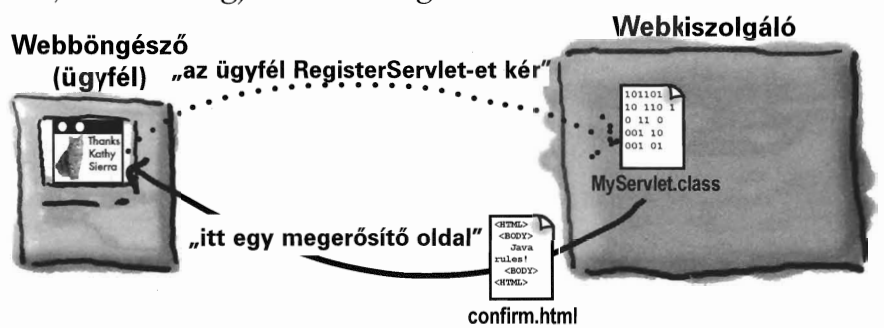
És a kiszolgálói kisalkalmazások is képesek használni az RMI-t!

A J2EE technológia messze leggyakoribb felhasználási módja a kiszolgálói kisalkalmazások és az EJB-k keverése, amikor is a servlet-ek az EJB ügyfelei. Ebben az esetben a *servlet RMI segítségével társalog az EJB-kkel.* (Bár az RMI EJB-kkel történő használata *kissé* különbözik az imént bemutatott eljárástól.)

1. Az ügyfél kitölt egy bejelentkezési űrlapot, és a „submit" gombra kattint. A HTTP-kiszolgáló (vagyis a webkiszolgáló) megkapja a kérelmet, és látva, hogy az egy kiszolgálói kisalkalmazásra irányul, átadja neki.



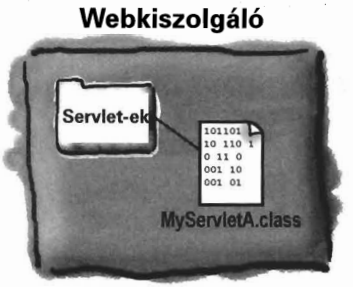
2. A kiszolgálói kisalkalmazás (Java-) kódja elindul, adatokat ad az adatbázishoz, összeállít egy weboldalt (egyedi információkkal), és ez utóbbit visszaküldi az ügyfélnek, ahol az megjelenik a böngészőben.



Kiszolgálói kisalkalmazás készítése és futtatása

① Derítsd ki, hová kell tenned a kiszolgálói kisalkalmazásaidat.

A példáinkban feltételezzük, hogy már üzembe helyeztél egy webkiszolgálót, és beállítottad a kiszolgálói kisalkalmazások támogatására. A legfontosabb dolog annak kiderítése, hogy pontosan hol kell elhelyezni a kiszolgálói kisalkalmazásod osztályfájljait ahhoz, hogy a kiszolgáló „lássa" azokat. Ha a webhelyednek egy internetszolgáltató ad helyet, a szolgáltatótól kell megkérdezned, hogy hová teheted a kiszolgálói kisalkalmazásaidat, ahogy tőlük kell megtudakolnod a CGI-parancsfájlok helyét is.



② Szerezd be a servlets.jar fájlt, és add hozzá az osztályelérési útvonalodhoz.

A kiszolgálói kisalkalmazások nem részei a Java alapkönyvtárainak; a működtetésükhöz a servlets.jar fájlba csomagol servlet-osztályokra lesz szükséged. A servlet-osztályokat letöltheted a java.sun.com címről, illetve beszerezheted őket egy Java-képes webböngésző részeként (amilyen például az Apache Tomcat, amelyet az apache.org webhelyen érhetsz el). Ezek nélkül az osztályok nélkül a kiszolgálói kisalkalmazásaidat nem fogod tudni lefordítani.


servlets.jar

③ Írj egy servlet-osztályt a HttpServlet bővítésével.

A kiszolgálói kisalkalmazások csupán Java-osztályok, amelyek a (javax.servlet.http csomagban található) HttpServlet osztályt bővítik. Más fajta kiszolgálói kisalkalmazásokat is készíthetsz, de a legtöbbször HttpServlet-ekre lesz szükséged.



MyServletA.class

```
public class MyServletA extends HttpServlet { ... }
```

④ Írj egy HTML-oldalt, amely meghívja a kiszolgálói kisalkalmazásodat.

Amikor a felhasználó egy olyan hivatkozásra kattint, ami a kiszolgálói kisalkalmazásodra mutat, a webkiszolgáló megkeresi a servlet-et, és HTTP-parancstól (GET, POST stb.) függően meghívja a megfelelő tagfüggvényt.

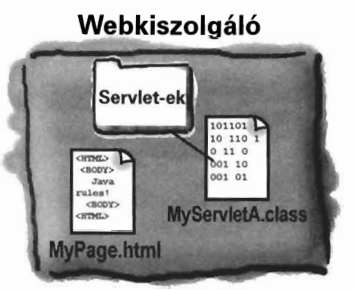


MyPage.html

```
<a href="servlets/MyServletA">Ez a legcsodálatosabb
                              kiszolgálói kisalkalmazás.</a>
```

⑤ Tedd a kiszolgálói kisalkalmazást és a HTML-oldalt elérhetővé a kiszolgáló számára.

Ez a lépés teljesen a webkiszolgálódtól függ (pontosabban attól, hogy a Java Servlets-nek melyik *változatát* használod). Lehet, hogy az internetszolgáltatód egyszerűen azt mondja, hogy dobd a kiszolgálói kisalkalmazást a webhelyed „Servlets" könyvtárába. Ha viszont mondjuk a Tomcat legfrissebb változatát használod, sokkal több dolgod lesz azzal, hogy a servlet-et (és a weboldalt) a megfelelő helyre juttasd. (Történetesen erről is írtunk egy könyvet.)



Egy nagyon egyszerű kiszolgálói kisalkalmazás

Az io mellett két servlet-csomagot is be kell töltened. Ne feledd, hogy ezek a csomagok NEM részei a Java alapkönyvtárainak, ezért külön kell letöltened őket.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

A legtöbb „szokványos" kiszolgálói kisalkalmazás a HttpServlet osztályt bővíti, majd felülbírálja annak egy vagy több tagfüggvényét.

```
public class MyServletA extends HttpServlet {
```

Az egyszerű GET HTTP-üzenetekhez felülírjuk a doGet tagfüggvényt.

A webkiszolgáló ezt a tagfüggvényt hívja meg, átadva neked az ügyfél kérelmét (amelyből adatokat nyerhetsz ki), valamint egy „response" (válasz) objektumot, amelyet a válasz (egy oldal) visszaküldésére használsz majd.

```
    public void doGet (HttpServletRequest request, HttpServletResponse response)
                                        throws  ServletException, IOException  {
```

Ez árulja el a kiszolgálónak (és a böngészőnek), hogy milyen „dolog" érkezik majd a kiszolgálótól az adott kiszolgálói kisalkalmazás futtatásának eredményeként.

```
        response.setContentType("text/html");
```

A válaszobjektum egy kimeneti adatfolyamot ad, amelybe információkat „írhatunk" vissza a kiszolgálónak.

```
        PrintWriter out = response.getWriter();

        String message = "Ha ezt olvasod, akkor működött!";

        out.println("<HTML><BODY>");
        out.println("<H1>" + message + "</H1>");
        out.println("</BODY></HTML>");
        out.close();
    }
}
```

Amit „kiírunk", az egy HTML-oldal! Az oldal a kiszolgálón keresztül jut vissza a böngészőhöz, mint bármely más HTML-oldal, annak ellenére, hogy ez az oldal eddig nem is létezett. Más szavakkal, nincs sehol olyan .html fájl, amelynek ez lenne a tartalma.

HTML-oldal a fenti kiszolgálói kisalkalmazásra mutató hivatkozással

A kiszolgálói kisalkalmazás elindításához erre a hivatkozásra kell kattintani.

```
<HTML>
  <BODY>
    <a href="servlets/MyServletA">This is an amazing servlet.</a>
  </BODY>
</HTML>
```

Így fest a weboldal:





Újratöltve

- A kiszolgálói kisalkalmazások (servlet) Java-osztályok, amelyek teljes egészében egy HTTP- (web-) kiszolgálón (illetve azon belül) futnak.
- A kiszolgálói kisalkalmazások azért hasznosak, mert a segítségükkel egy weboldalon végrehajtott ügyfélművelet eredményeként a kiszolgálón hajthatsz végre kódot. Ha egy ügyfél például egy webes űrlapon keresztül információkat ad meg, a kiszolgálói kisalkalmazás feldolgozhatja azokat, hozzáadhatja egy adatbázishoz, és visszaküldhet egy testreszabott, megerősítő válaszoldalt.
- A kiszolgálói kisalkalmazások lefordításához a servlets.jar fájlban található servlet-csomagokra van szükség. A servlet-osztályok nem részei a Java alapkönyvtárainak, ezért a java.sun.com címről vagy egy kiszolgálói kisalkalmazások futtatására képes webkiszolgálóval együtt kell letöltened őket. (Megjegyzés: a Java 2 Enterprise Edition – J2EE – tartalmazza a Servlet könyvtárat.)
- Egy kiszolgálói kisalkalmazás futtatásához mindenképpen rendelkezned kell egy erre képes webkiszolgálóval, például az apache.org címről beszerezhető Tomcattel.
- A kiszolgálói kisalkalmazásaidat az adott webkiszolgálóra jellemző helyen kell elhelyezned, ezért ezt előbb ki kell derítened, mielőtt futtathatnád őket. Ha a webhelyednek egy kiszolgálói kisalkalmazásokat támogató internetszolgáltató (ISP) ad helyet, az ISP-től kell megtudakolnod, hogy melyik könyvtárba teheted a kiszolgálói kisalkalmazásokat.
- A kiszolgálói kisalkalmazások jellemzően a HttpServlet osztályt bővítik, felülbírálva annak egy vagy több tagfüggvényét, például a doGet() vagy a doPost() tagfüggvényt.
- A webkiszolgáló az ügyfél kérelme alapján indítja el a kiszolgálói kisalkalmazást, és hívja meg a megfelelő tagfüggvényt (doGet() stb.).
- A kiszolgálói kisalkalmazás a doGet() tagfüggvény válaszparaméteréből nyert PrintWriter kimeneti adatfolyamon keresztül küldhet vissza egy választ.
- A kiszolgálói kisalkalmazás egy HTML-oldalt „ír ki", a megfelelő HTML-címkékkel együtt.

Nincsenek

hülye kérdések

K: Mi az a JSP, és mi köze a kiszolgálói kisalkalmazásokhoz?

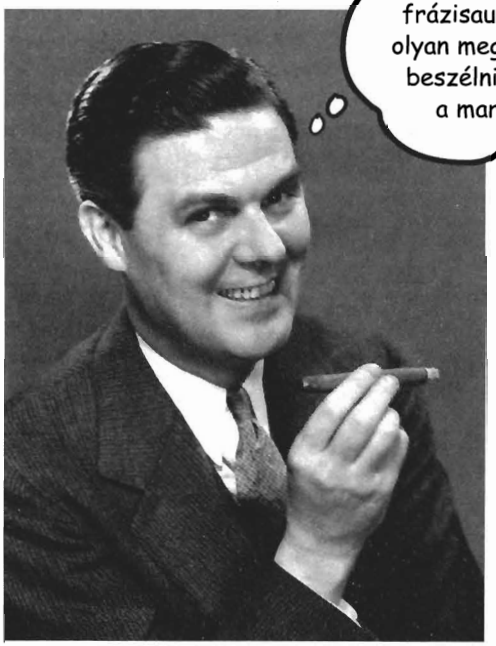
V: A JSP a Java Server Pages (Java kiszolgálói oldalak) rövidítése.

A webkiszolgáló a JSP-ket végül kiszolgálói kisalkalmazásokká alakítja – a különbséget egy servlet és egy JSP között az jelenti, hogy TE (a fejlesztő) mit hozol létre. A kiszolgálói kisalkalmazások esetében egy Java-*osztályt* írsz, amely a kiíró utasításokban tartalmaz HTML-t (amennyiben egy HTML-oldalt küldesz vissza az ügyfélnek), míg a JSP-knél ennek éppen az ellenkezőjét teszed: egy *HTML*-oldalt készítesz, amely *Java*-kódot tartalmaz! Ez lehetővé teszi, hogy dinamikus weboldalakat hozz létre, amikor is az oldalt szokványos HTML-oldalként írod meg, csak éppen Java-kódot ágyazol bele (illetve egyéb címkéket, amelyek futásidőben „elindítják" a Java-kódot), amelynek a feldolgozására futáskor kerül sor. Más szavakkal, az oldal egy része a Java-kód lefutásakor testreszabott formát ölt. A JSP fő előnye a szokványos kiszolgálói kisalkalmazásokkal szemben az, hogy sokkal egyszerűbb egy servlet HTML-részét JSP-oldalként megírni, mint fáradságos munkával a servlet válaszában kiíró utasításokkal előállítani a HTML-t. Képzelj el egy átlagosan bonyolult HTML-oldalt, és képzeld el, hogy println utasításokkal kell formáznod. Fúj! Sok alkalmazásban azonban nem szükséges JSP-ket használni, mert a servlet-nek nem kell dinamikus válaszokat visszaküldenie, vagy mert a HTML elég egyszerű ahhoz, hogy ez ne jelentsen különösebb gondot. Ezenkívül még mindig sok olyan webkiszolgáló van, amely támogatja a kiszolgálói kisalkalmazásokat, de a JSP-ket nem. A JSP-k másik előnye, hogy a munka megoszthatóː a Java-fejlesztők megírják a kiszolgálói kisalkalmazásokat, a webfejlesztők pedig a JSP-ket. Legalábbis a JSP ezt ígéri. A valóságban a JSP-k készítéséhez meg kell tanulni a Java használatát (és persze a HTML-címkékét is), vagyis reálisan nem várható el, hogy egy HTML-oldaltervező a kisujjából kirázzon egy JSP-t – legalábbis segédeszközök nélkül nem. Jó hír azonban, hogy egyre több szerzői segédeszköz jelenik meg, amelyekkel a weboldal-tervezők anélkül hozhatnak létre JSP-ket, hogy az alapoktól kezdve kódot kellene írniuk.

K: Csak ennyi a mondanivalótok a kiszolgálói kisalkalmazásokról? Miután *ekkora* hűhót csaptatok az RMI körül?

V: Igen. Az RMI a Java nyelv része, és az RMI minden osztálya megtalálható az alapkönyvtárakban. A kiszolgálói kisalkalmazások és a JSP-k ezzel szemben *nem* részei a Javának, hanem *szabványos bővítményeknek* számítanak. Az RMI-t bármely modern JVM-en futtathatod, a kiszolgálói kisalkalmazások és JSP-k azonban egy megfelelően beállított webkiszolgálót igényelnek, amely rendelkezik egy „servlet-tárolóval". Röviden: a téma „túlmutat könyvünk keretein" – a sorozat *Head First Servlets & JSP* című kötetéből azonban többet is megtudhatsz róla.

Csak a móka kedvéért: alakítsd kiszolgálói kisalkalmazássá a Frázisautomatát!



Most, hogy közöltük veled, hogy nincs több mondanivalónk a kiszolgálói kisalkalmazásokról, nem tudunk ellenállni a kísértésnek, hogy „szervletizáljuk" az 1. fejezetben bemutatott Frázisautomatát. A kiszolgálói kisalkalmazások is csak Java-kódok, és egy Java-kód meghívhatja más osztályok Java-kódját, tehát egy kiszolgálói kisalkalmazás is nyugodtan meghívhatja a Frázisautomata valamelyik tagfüggvényét. Csak annyit kell tenned, hogy a Frázisautomata osztályát ugyanabba a könyvtárba teszed, ahol a kiszolgálói kisalkalmazás is található, és máris megtetted az első lépést. (A Frázisautomata kódját a következő oldalon láthatod.)

```
import java.io.*;

import javax.servlet.*;
import javax.servlet.http.*;

public class KathyServlet extends HttpServlet {
    public void doGet (HttpServletRequest request, HttpServletResponse response)
                                        throws ServletException, IOException  {

        String title = "PhraseOMatic has generated the following phrase.";

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.println("<HTML><HEAD><TITLE>");
        out.println("PhraseOmatic");
        out.println("</TITLE></HEAD><BODY>");
        out.println("<H1>" + title + "</H1>");
        out.println("<P>" + PhraseOMatic.makePhrase());
        out.println("<P><a href=\"KathyServlet\">make another phrase</a></p>");
        out.println("</BODY></HTML>");

        out.close();
    }
}
```

Látod? A kiszolgálói kisalkalmazásod meghívhatja egy másik osztály tagfüggvényeit. Ebben az esetben a PhraseOMatic osztály (lásd a következő oldalon) statikus makePhrase() tagfüggvényét hívjuk.

A servlet-barát Frázisautomata kódja

A kódnak ez a változata kissé különbözik attól, amit az 1. fejezetben láttál. Az eredeti változatban a teljes cuccot a main() tagfüggvényben futtattuk, és a programot minden alkalommal újra kellett futtatnunk, amikor új frázist szerettünk volna megjeleníttetni a parancssorban. Ebben a változatban a kód egyszerűen visszad egy karakterláncot (a frázissal), amikor meghívjuk a statikus makePhrase() tagfüggvényt. Így tetszőleges más kódból meghívhatod a tagfüggvényt, és visszakaphatsz egy karakterláncot a véletlenszerűen összeállított blablával.

Megjegyzendő, hogy a lenti hosszú String[] tömb-hozzárendelések tördelése a szövegszerkesztés eredménye – tehát a kötőjeleket ne írd be! Csak írj folyamatosan, és hagyd, hogy a kódszerkesztő végezze a tördelést. És bármit is teszel, ne nyomd le az ENTER/RETURN billentyűt egy karakterlánc közepén (vagyis amikor egy idézőjelen belül állsz).

```
public class PhraseOMatic {
   public static String makePhrase() {
      // Hozz létre három halmazt, amiből szavakat, illetve kifejezéseket választhatsz,
      // és add hozzá a saját szavaidat!
      String[] wordListOne = {"24/7","multi-Tier","30,000 foot","B-to-B","win-win","front-
end", "web-based","pervasive", "smart", "six-sigma","critical-path", "dynamic"};

      String[] wordListTwo = {"empowered", "sticky", "valued-added", "oriented", "centric",
"distributed", "clustered", "branded","outside-the-box", "positioned", "networked", "fo-
cused", "leveraged", "aligned", "targeted", "shared", "cooperative", "accelerated"};

      String[] wordListThree = {"process", "tipping point", "solution", "architecture",
"core competency", "strategy", "mindshare", "portal", "space", "vision", "paradigm", "mis-
sion"};

      // Derítsd ki, hány szó, illetve kifejezés van az egyes halmazokban!
      int oneLength = wordListOne.length;
      int twoLength = wordListTwo.length;
      int threeLength = wordListThree.length;

      // Állíts elő három véletlenszámot!
      int rand1 = (int) (Math.random() * oneLength);
      int rand2 = (int) (Math.random() * twoLength);
      int rand3 = (int) (Math.random() * threeLength);

      // Most építs fel egy szófordulatot!
      String phrase = wordListOne[rand1] + " " + wordListTwo[rand2] + " " +
wordListThree[rand3];

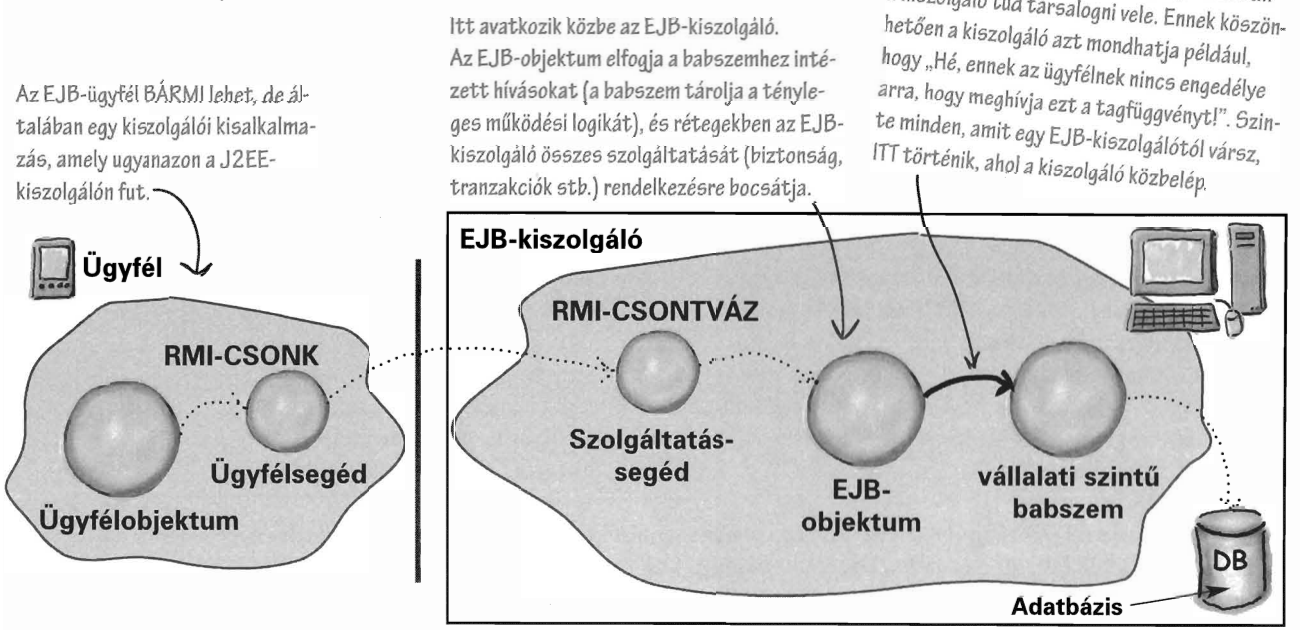
      // Írd ki a blablát!
      return ("What we need is a " + phrase);
   }
}
```

Enterprise JavaBeans: a felpumpált RMI

Az RMI kiválóan alkalmas arra, hogy távoli szolgáltatásokat írj és futtass. Önmagában az RMI-vel azonban nem tudnál egy olyan szolgáltatást működtetni, mint az Amazon vagy az eBay. Egy nagy, halálosan komoly, vállalati szintű („enterprise") alkalmazáshoz valami többre van szükség. Olyasmire, ami képes megbirkózni a tranzakciókkal, a nagy mennyiségű párhuzamos művelettel (mint amikor milliók bombázzák a kiszolgálódat egyszerre, hogy organikus kutyatápot vegyenek), a biztonsági kérdésekkel (az alkalmazottak fizetési adatbázisát például nem szabad bárkinek elérnie) vagy az adatkezeléssel. Ehhez egy *vállalati szintű alkalmazáskiszolgáló* szükséges. A Javában ez egy Java 2 Enterprise Edition (J2EE) kiszolgálót jelent. A J2EE kiszolgálóknak egyaránt része egy webkiszolgáló és egy Enterprise JavaBeans- (EJB-) – „vállalati szintű Java-babszem" – kiszolgáló, így lehetővé teszik, hogy kiszolgálói kisalkalmazásokat és EJB-ket egyaránt tartalmazó alkalmazásokat működtess. A kiszolgálói kisalkalmazásokhoz hasonlóan az EJB is túlmutat könyvünk keretein, és nincs rá mód, hogy akár csak egyetlen „apró" EJB-példát is bemutassunk a kódjával együtt, de egy gyors pillantást azért vetünk a működésére. (Az EJB részletes tárgyalása megtalálható a *Head First EJB* című kötetben, amelyet melegen ajánlunk a téma iránt komolyabban érdeklődőknek.)

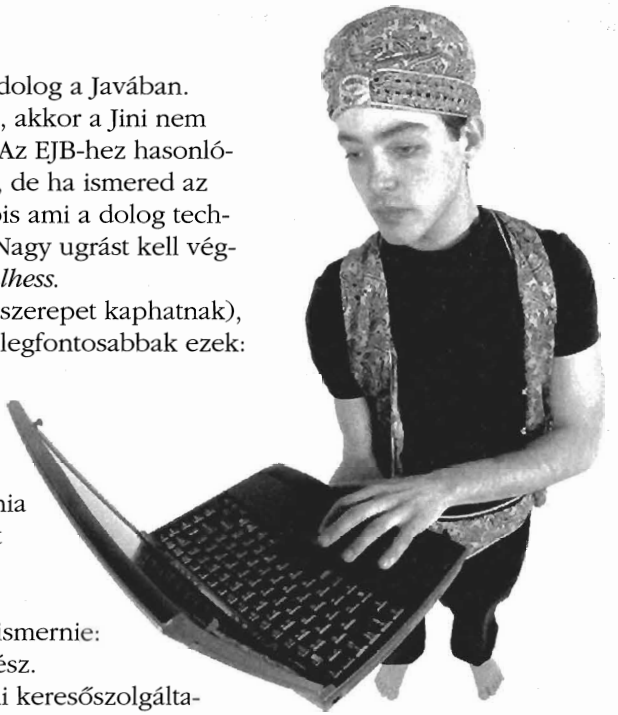
Az EJB-kiszolgálók egy csomó olyan szolgáltatást nyújtanak, amilyeneket az egyszerű RMI nem. Képesek például kezelni a tranzakciókat, a biztonsági kérdéseket, a párhuzamos hozzáférést, az adatbázisokat és a hálózatokat.

Az EJB-kiszolgáló egy RMI-hívás közben lép működésbe, és rétegekben az összes szolgáltatását rendelkezésre bocsátja.



Ez csak egy kis része az EJB-nek!

A végső trükk – egy kis Jini



Imádjuk a Jinit. Úgy gondoljuk, hogy a Jini a legjobb dolog a Javában. Ha az EJB egy felpumpált RMI (egy csomó kezelővel), akkor a Jini nem más, mint RMI *szárnyakkal*. Színtiszta Java-varázslat. Az EJB-hez hasonlóan a Jini világában sem merülhetünk el itt részletesen, de ha ismered az RMI-t, az út háromnegyedét már megtetted – legalábbis ami a dolog technikai részét illeti. Az *agyad* átállítása már más tészta. Nagy ugrást kell végrehajtanod: el kell rugaszkodnod a talajtól, hogy *repülhess*.

A Jini az RMI-re támaszkodik (bár más protokollok is szerepet kaphatnak), de néhány kulcsfontosságú új szolgáltatást is nyújt. A legfontosabbak ezek:

Alkalmazkodó felderítés

Öngyógyító hálózatok

Emlékezz vissza, hogy az RMI használatakor az ügyfélnek tudnia kell a távoi szolgáltatás nevét és helyét. A keresést végrehajtó ügyfélkódnak tartalmaznia kell a távoli szolgáltatás IP-címét vagy gépnevét (mert ott fut az RMI-jegyzék) *és* a szolgáltatás logikai nevét, amelyen azt bejegyezték.

A Jiniben azonban az ügyfélnek csak egy dolgot kell ismernie: a *szolgáltatás által megvalósított* ***felületet***! Ennyi az egész.

De hogyan találhatod meg, amit keresel? A trükk a Jini keresőszolgáltatásában rejlik, ami jóval sokoldalúbb és rugalmasabb, mint az RMI-jegyzék.

Először is, a Jini keresőszolgáltatása *automatikusan* bejelenti magát a hálózaton: amikor csatlakozik a hálózathoz, (IP-üzenetszórással) üzenetet küld, amelyben azt mondja: „Itt vagyok, ha érdekel valakit.".

De ez még nem minden. Tegyük fel, hogy te (egy ügyfél) az *után* csatlakozol a hálózatra, hogy a keresőszolgáltatás bejelentette magát. Ekkor *te* küldhetsz üzenetet az egész hálózatnak, amelyben megkérdezed, hogy van-e elérhető keresőszolgáltatás.

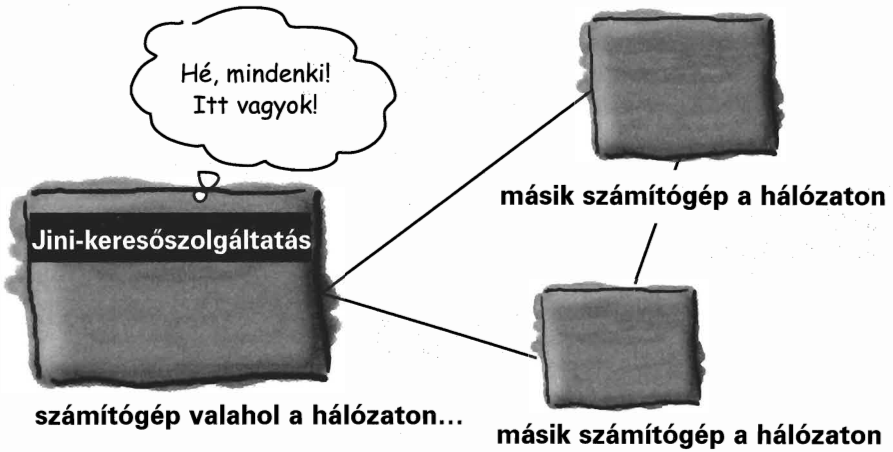
Persze téged valójában nem *maga* a keresőszolgáltatás érdekel, hanem azok a szolgáltatások, amelyeket bejegyeztek a keresőszolgáltatásba: távoli RMI-szolgáltatások, egyéb sorosítható Java-objektumok, vagy akár olyan eszközök, mint a nyomtatók, kamerák és kávéautomaták.

Itt válik a dolog igazán izgalmassá: amikor egy szolgáltatás csatlakozik a hálózatra, dinamikusan felderíti a hálózaton található Jini-keresőszolgáltatásokat (és be is jegyezteti magát velük). Amikor a szolgáltatás feliratkozik a keresőszolgáltatásnál, egy sorosított objektumot küld neki, amit a keresőszolgáltatásban kell elhelyezni. Ez a sorosított objektum lehet egy távoli RMI-szolgáltatás csonkja, egy hálózati eszköz illesztőprogramja, sőt akár a teljes szolgáltatás is, ami (miután megszerezted a keresőszolgáltatástól) helyben, a te számítógépeden fut. A szolgáltatás pedig nem *név*, hanem az általa megvalósított *felület* szerint jegyezteti be magát.

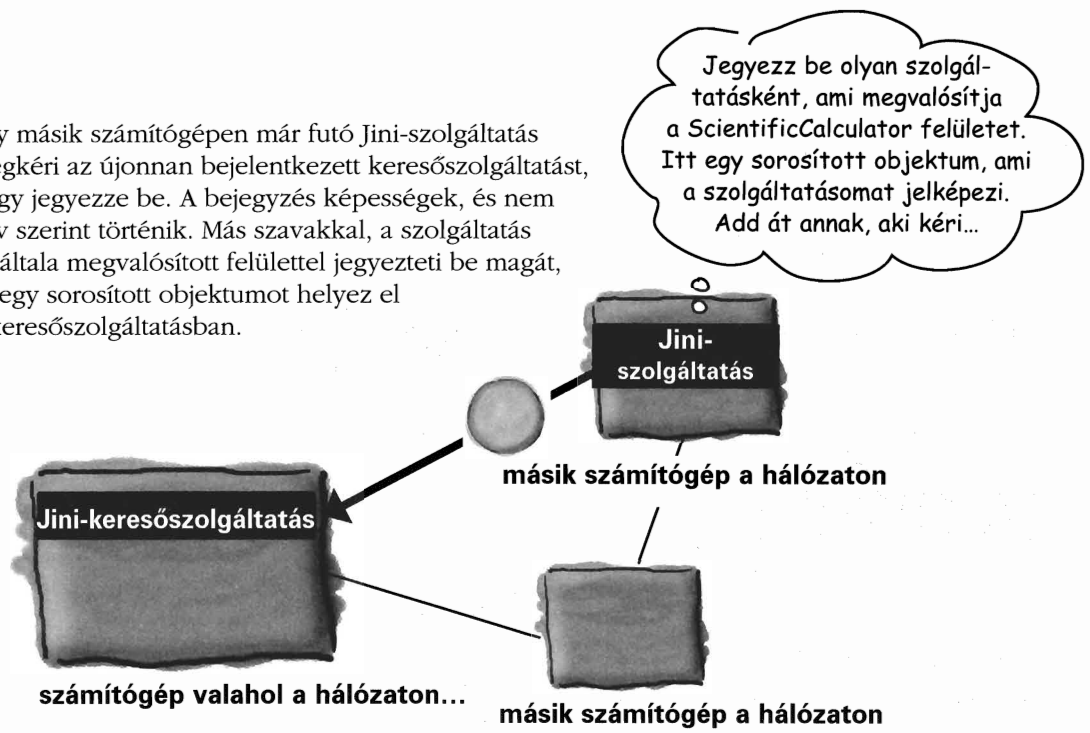
Miután te (az ügyfél) hozzájutottál egy hivatkozáshoz, ami a keresőszolgáltatásra mutat, ilyen kérdéseket tehetsz fel a keresőnek: „Hé, van valamid, ami megvalósítja a ScientificCalculator felületet?". A keresőszolgáltatás ekkor megnézi a listáját a bejegyzett felületekről, és ha talál egyezést, azt válaszolja, hogy „Igen, *van* valamim, ami megvalósítja a kért felületet. Tessék a sorosított objektum, amit a ScientificCalculator szolgáltatás bejegyzett."

Az alkalmazkodó felderítés működés közben

1. A Jini-keresőszolgáltatás elindul valahol a hálózaton, és IP-üzenetszórással bejelenti magát.

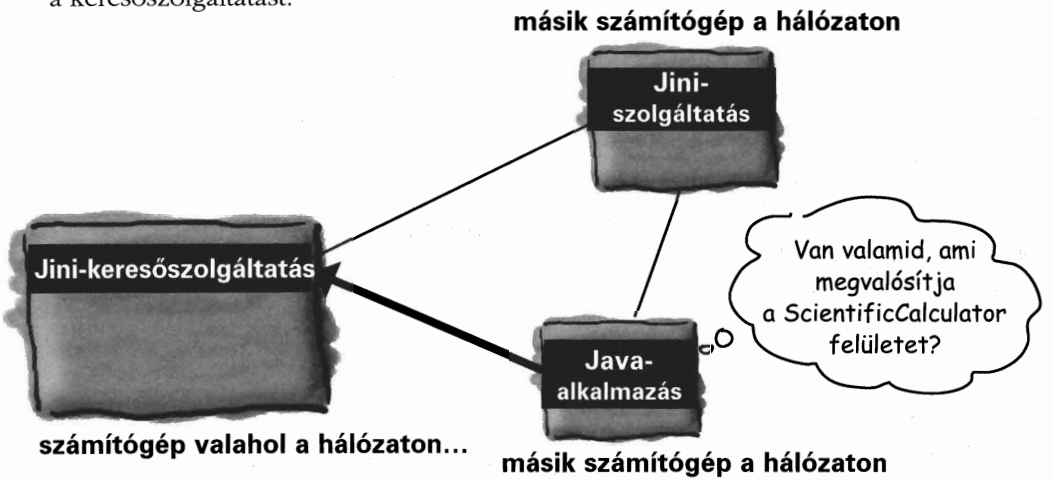


2. Egy másik számítógépen már futó Jini-szolgáltatás megkéri az újonnan bejelentkezett keresőszolgáltatást, hogy jegyezze be. A bejegyzés képességek, és nem név szerint történik. Más szavakkal, a szolgáltatás az általa megvalósított felülettel jegyezteti be magát, és egy sorosított objektumot helyez el a keresőszolgáltatásban.

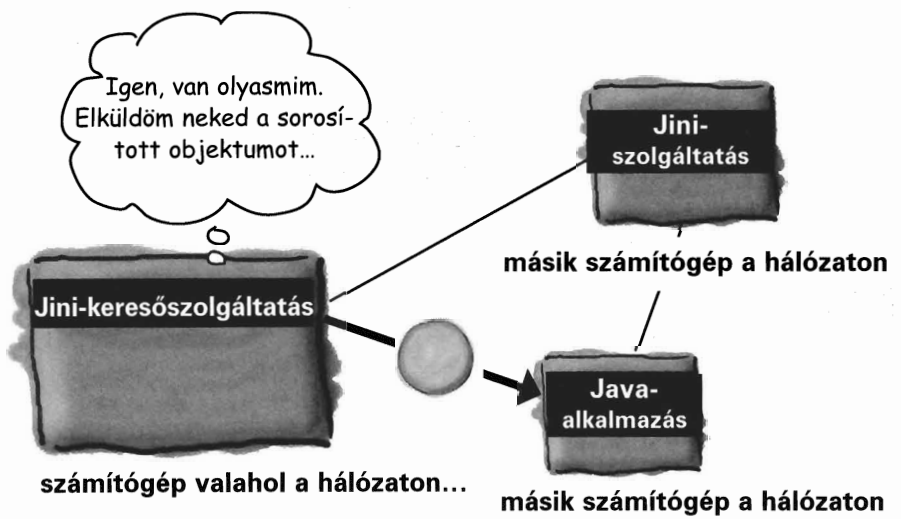


Az alkalmazkodó felderítés működés közben – folytatás

3 Egy ügyfél a hálózaton valami olyasmit szeretne, ami megvalósítja a ScientificCalculator felületet. Fogalma sincs, hogy létezik-e ilyen (és ha igen, akkor hol), ezért megkérdezi a keresőszolgáltatást.

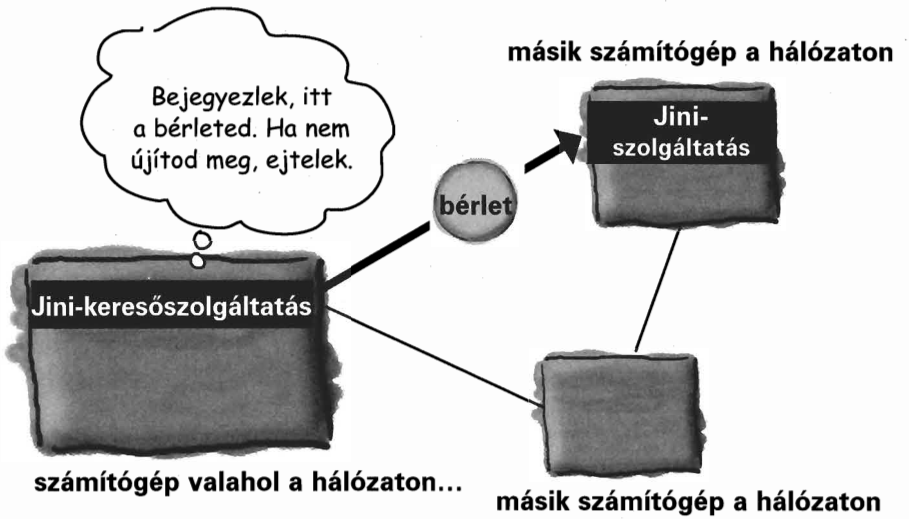


4 A keresőszolgáltatás válaszol, mivel bejegyeztek nála valamit ScientificCalculator felületként.

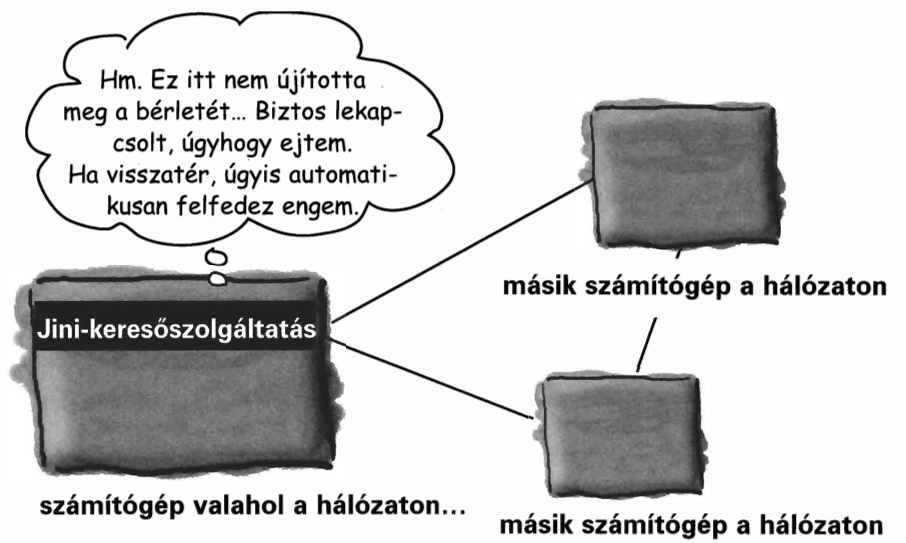


Öngyógyító hálózat működés közben

1. Egy Jini-szolgáltatás bejegyzést kér a keresőszolgáltatástól. A keresőszolgáltatás egy „bérlettel" (lease) válaszol, amit az újonnan bejegyzett szolgáltatásnak folyamatosan meg kell újítania, különben a keresőszolgáltatás feltételezi, hogy a szolgáltatás levált a hálózatról. A keresőszolgáltatás mindig pontos képet akar adni a hálózat többi részének az elérhető szolgáltatásokról.

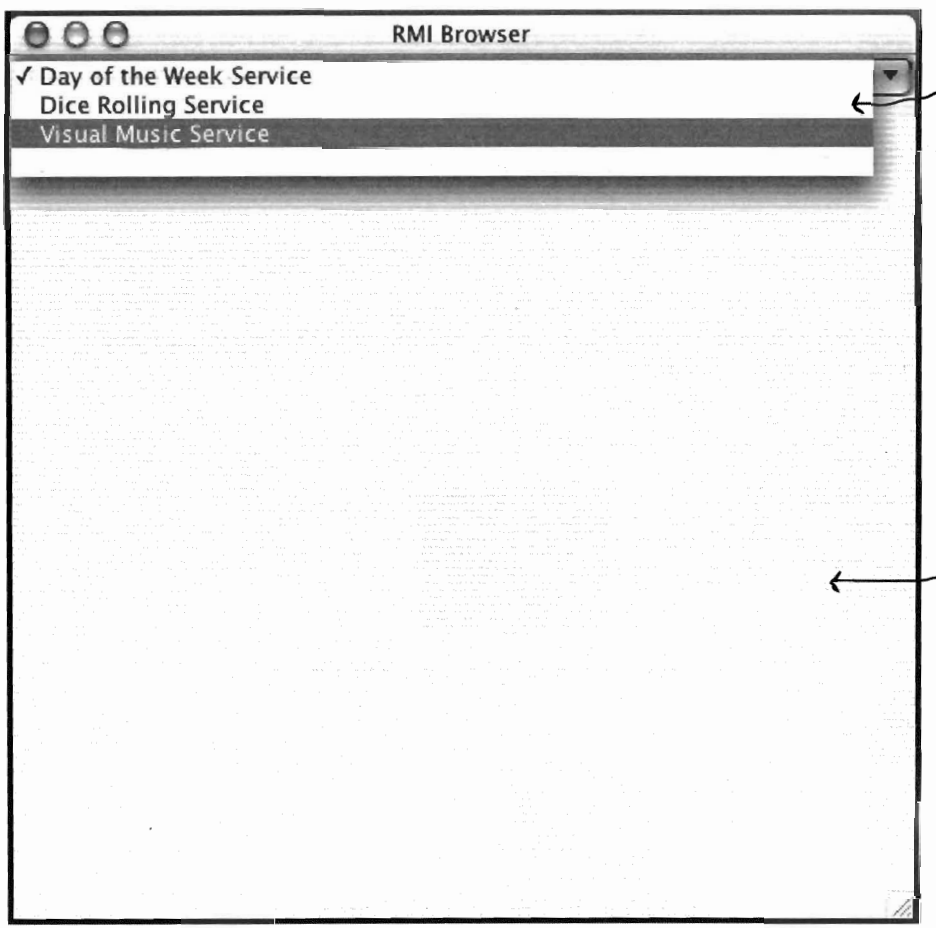


2. A szolgáltatás leválik a hálózatról (mert valaki lekapcsolja), ezért nem újítja meg a bérletét a keresőszolgáltatásnál, így a keresőszolgáltatás ejti.



Az utolsó projekt: az univerzális szolgáltatásböngésző

Valami olyasmit fogunk elkészíteni, ami nem Jini-képes, de könnyen azzá tehető. Ízelítőt ad a Jini képességeiből, de tiszta RMI használatával. Valójában a fő különbség a mi programunk és egy Jini-alkalmazás között az, ahogy a szolgáltatás felderítése történik. A Jini-keresőszolgáltatás helyett, amely automatikusan bejelenti magát, és a hálózaton bárhol lehet, az RMI-jegyzéket használjuk, amelynek a távoli szolgáltatással megegyező gépen kell lennie, és amely nem reklámozza magát önműködően. Ezenkívül a szolgáltatásunk nem jegyezteti be magát automatikusan a keresőszolgáltatással, hanem *nekünk* kell felvennünk az RMI-jegyzékbe (a Naming.rebind() tagfüggvénnyel). Miután azonban az ügyfél megtalálta a szolgáltatást az RMI-jegyzékben, az alkalmazás többi része majdnem pontosan ugyanolyan lesz, mint ahogy a Jiniben csinálnánk. (A legfontosabb dolog, ami hiányzik, a *bérlet*, ami lehetővé tenné, hogy a hálózat meggyógyítsa önmagát, ha valamelyik szolgáltatás kiesik.)
Az univerzális szolgáltatásböngésző olyan, mint egy különleges célú webböngésző, amely HTML-oldalak helyett interaktív Java GUI-kat tölt le és jelenít meg, amelyeket *univerzális szolgáltatásoknak* nevezünk.



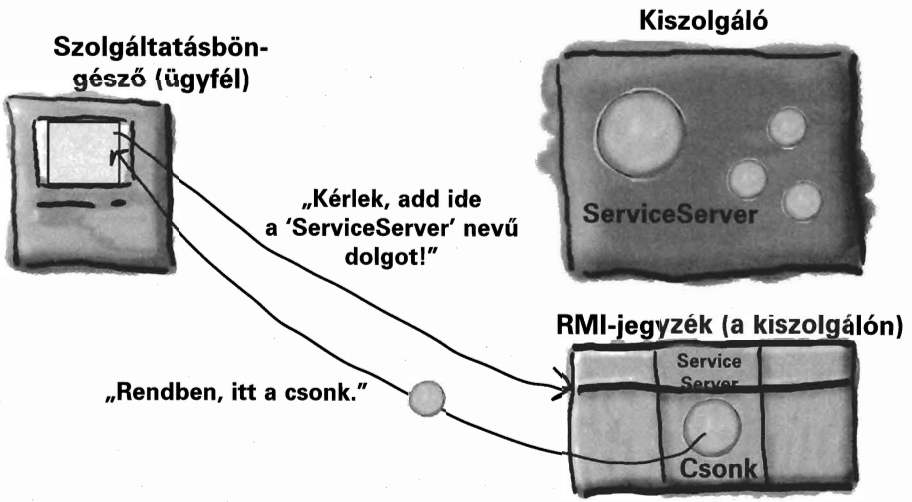
Válassz egy szolgáltatást a listából. A távoli RMI-szolgáltatásnak van egy getServiceList() nevű tagfüggvénye, amely ezt a szolgáltatáslistát adja vissza.

Amikor a felhasználó kiválaszt egy szolgáltatást, az ügyfél elkéri a tényleges szolgáltatást (DiceRolling, DayOfTheWeek stb.) a távoli RMI-szolgáltatástól.

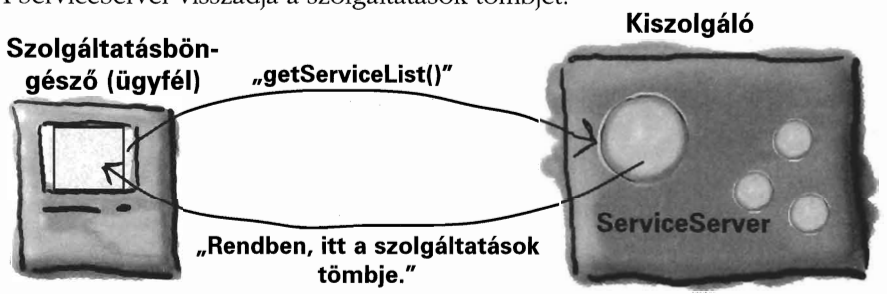
Amikor kiválasztasz egy szolgáltatást, az itt fog megjelenni.

Így működik:

1. Az ügyfél elindul, keresést hajt végre az RMI-jegyzékben a „ServiceServer" szolgáltatás után, és visszakapja a csonkot.



2. Az ügyfél meghívja a csonk getServiceList() tagfüggvényét. A ServiceServer visszadja a szolgáltatások tömbjét.

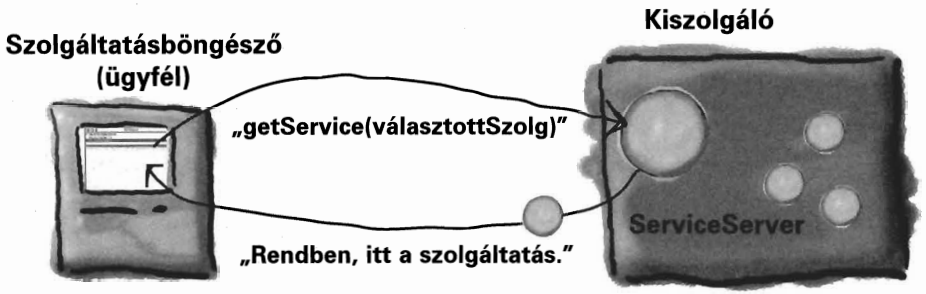


3. Az ügyfél megjeleníti a szolgáltatások listáját egy grafikus felületen.



Így működik – folytatás...

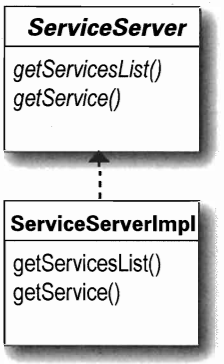
4. A felhasználó kiválaszt egy szolgáltatást a listából, ezért az ügyfél meghívja a getService() tagfüggvényt a távoli szolgáltatásra. A távoli szolgáltatás egy sorosított objektumot ad vissza, ami egy tényleges szolgáltatás, ami az ügyfélböngésző belsejében fog futni.



5. Az ügyfél meghívja a távoli szolgáltatástól az imént kapott sorosított szolgáltatásobjektum getGuiPanel() tagfüggvényét. Az említett szolgáltatás grafikus felületet megjelenik a böngészőben, és a felhasználó helyben műveleteket végezhet rajta. Ezen a ponton nincs szükség a távoli szolgáltásra, amíg a felhasználó úgy nem dönt, hogy másik szolgáltatást választ.



A szükséges osztályok és felületek:

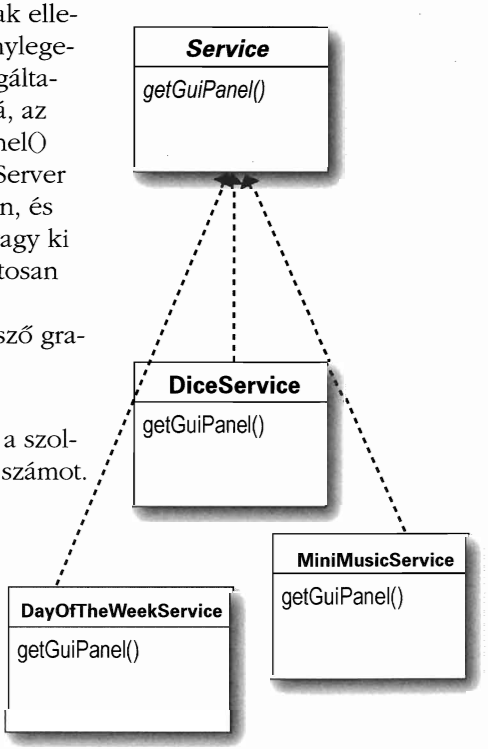


1 interface ServiceServer implements Remote
Egy szokványos távoli RMI-felület a távoli szolgáltatáshoz (a szolgáltatáslista lekérésére és a kiválasztott szolgáltatás visszaadására szolgáló tagfüggvénnyel a távoli szolgáltatás rendelkezik).

2 class ServiceServerImpl implements ServiceServer
A tényleges távoli RMI-szolgáltatás (ami az UnicastRemoteObject osztályt bővíti). A feladata a szolgáltatások (az ügyfélnek átadott dolgok) példányosítása és tárolása és magának a kiszolgálónak (ServiceServerImpl) a bejegyzése az RMI-jegyzékbe.

3 class ServiceBrowser
Az ügyfél. Egy nagyon egyszerű grafikus felhasználói felületet épít fel, keresést végez az RMI-jegyzékben, hogy megszerezze a ServiceServer csonkot, majd meghív rá egy távoli tagfüggvényt, hogy kiolvassa és megjelenítse a szolgáltatáslistát a grafikus felületen.

4 interface Service
Ez a kulcsa az egésznek. Ez az igen egyszerű felület csupán egy tagfüggvénnyel – getGuiPanel() – rendelkezik. Az ügyfélnek átadott minden szolgáltatásnak meg kell valósítania ezt a felületet. Ez teszi az egészet UNIVERZÁLISSÁ. Ennek a felületnek a megvalósításával az ügyfél annak ellenére hozzájuthat egy szolgáltatáshoz, hogy fogalma sincs, hogy ténylegesen milyen osztály (vagy osztályok) alkotja (alkotják) az adott szolgáltatást. Az ügyfél mindössze annyit tud, hogy bármi is érkezzen hozzá, az megvalósítja a Service felületet, tehát rendelkeznie KELL getGuiPanel() tagfüggvénnyel. Az ügyfél egy sorosított objektumot kap a ServiceServer csonkra kiadott getService(*választottSzolg*) hívás eredményeképpen, és csak annyit mond ennek az objektumnak, hogy „Nem tudom, mi vagy ki vagy, de azt IGEN, hogy megvalósítod a Service felületet, ezért biztosan meghívhatom rád a getGuiPanel() tagfüggvényt. Mivel pedig a getGuiPanel() egy JPanel-t ad vissza, csak lepottyantom a böngésző grafikus felületére, és máris műveleteket végezhetek rajta!".



5 class DiceService implements Service
Van dobókockád (dice)? Ha nincs, de dobni szeretnél, használd ezt a szolgáltatást, és egy virtuális dobókockával dobhatsz egy 1 és 6 közötti számot.

6 class MiniMusicService implements Service
Emlékszel még a fantasztikus kis „zenés videóprogramra" az első Kódkonyhából? Nos, ezt a programot szolgáltatássá alakítottuk: újra és újra lejátszhatod a „videót", amíg a szobatársaid menekülőre nem fogják.

7 class DayOfTheWeekService implements Service
Pénteki napon születtél? Írd be a születésnapod dátumát, és derítsd ki!

interface ServiceServer (a távoli felület)

```
import java.rmi.*;
public interface ServiceServer extends Remote {
    Object[] getServiceList() throws RemoteException;
    Service getService(Object serviceKey) throws RemoteException;
}
```

Szokványos távoli RMI-felület, amely a távoli szolgáltatás két tagfüggvényét határozza meg.

interface Service (ezt valósítják meg a grafikus felületű szolgáltatások)

```
import javax.swing.*;
import java.io.*;

public interface Service extends Serializable {
    public JPanel getGuiPanel();
}
```

Hagyományos (értsd: nem távoli) felület, amely azt az egy tagfüggvényt – getGuiPanel() – határozza meg, amellyel minden univerzális szolgáltatásnak rendelkeznie kell. A felület a Serializable osztályt bővíti, így a Service felületet megvalósító bármely osztály automatikusan sorosítható lesz.

Ez elengedhetetlen, mert a szolgáltatásokat át kell vinni a hálózaton a kiszolgálóról, amikor az ügyfél meghívja a getService() tagfüggvényt a távoli ServiceServer-re.

class ServiceServerImpl (a távoli megvalósítás)

```
import java.rmi.*;
import java.util.*;
import java.rmi.server.*;
```

Szokványos RMI-megvalósítás

```
public class ServiceServerImpl extends UnicastRemoteObject implements ServiceServer  {

    HashMap serviceList;
```

A szolgáltatások egy HashMap gyűjteményben fognak tárolódni. EGY helyett KÉT objektumot teszünk a gyűjteménybe – egy kulcsobjektumot (például egy karakterláncot) és egy értékobjektumot (ez tetszőleges lehet). (A HashMap-ről a B függelék szól részletesebben.)

```
    public ServiceServerImpl() throws RemoteException {
        setUpServices();
    }
```

Amikor a konstruktor meghívódik, előkészítjük a tényleges univerzális szolgáltatásokat (DiceService, MiniMusicService stb.).

```
    private void setUpServices() {
        serviceList = new HashMap();
        serviceList.put("Dice Rolling Service", new DiceService());
        serviceList.put("Day of the Week Service", new DayOfTheWeekService());
        serviceList.put("Visual Music Service", new MiniMusicService());
    }
```

Létrehozzuk a szolgáltatásokat (a tényleges szolgáltatásobjektumokat), és beletesszük őket a HashMap-be, kulcsként egy String formátumú nevet megadva.

```
    public Object[] getServiceList() {
        System.out.println("in remote");
        return serviceList.keySet().toArray();
    }
```

Az ügyfél azért hívja meg ezt, hogy hozzájusson a szolgáltatások listájához, amelyet megjelenít a böngészőben (hogy a felhasználó kiválaszthasson egyet). A HashMap-ben levő KULCSOKBÓL tömböt létrehozva egy Object típusú tömböt adunk át (bár String-ek vannak benne). A tényleges Service objektumot nem küldjük el, csak ha az ügyfél kéri a getServic() meghívásával.

```
    public Service getService(Object serviceKey) throws RemoteException {
        Service theService = (Service) serviceList.get(serviceKey);
        return theService;
    }
```

Az ügyfél azt követően hívja meg ezt a tagfüggvényt, hogy a felhasználó kiválasztott egy szolgáltatást a megjelenített szolgáltatáslistából (amelyhez a fenti tagfüggvényből jutott hozzá). Ez a kód a kulcsot (ugyanazt a kulcsot, mint amit eredetileg elküldtünk az ügyfélnek) használja, hogy kinyerje a megfelelő szolgáltatást a HashMap-ből.

```
    public static void main (String[] args) {
        try {
          Naming.rebind("ServiceServer", new ServiceServerImpl());
        } catch(Exception ex) {
            ex.printStackTrace();
        }
        System.out.println("Remote service is running");
    }
```

class ServiceBrowser (az ügyfél)

```
import java.awt.*;
import javax.swing.*;
import java.rmi.*;
import java.awt.event.*;


public class ServiceBrowser {

   JPanel mainPanel;
   JComboBox serviceList;
   ServiceServer server;


   public void buildGUI() {
      JFrame frame = new JFrame("RMI Browser");
      mainPanel = new JPanel();
      frame.getContentPane().add(BorderLayout.CENTER, mainPanel);

      Object[] services = getServicesList();

      serviceList = new JComboBox(services);

      frame.getContentPane().add(BorderLayout.NORTH, serviceList);

      serviceList.addActionListener(new MyListListener());

      frame.setSize(500,500);
      frame.setVisible(true);

  }

   void loadService(Object serviceSelection) {
       try {
          Service svc = server.getService(serviceSelection);

          mainPanel.removeAll();
          mainPanel.add(svc.getGuiPanel());
          mainPanel.validate();
          mainPanel.repaint();
        } catch(Exception ex) {
          ex.printStackTrace();
        }
   }
```

Ez a tagfüggvény hajtja végre a keresést az RMI-jegyzékben, szerzi meg a csonkot, és hívja meg a getServiceList() tagfüggvényt.

Hozzáadjuk a szolgáltatásokat (egy Object-tömböt) a JComboBox-hoz (a listához). A JComboBox tudja, hogyan hozhat létre megjeleníthető String-eket a tömb elemeiből.

Itt adjuk hozzá a tényleges szolgáltatást a grafikus felülethez, miután a felhasználó kiválasztott egyet. (Ezt a tagfüggvényt a JComboBox eseményfigyelője hívja meg.) Meghívjuk a getService()-t a távoli kiszolgálóra (a ServiceServer csonkjára), és átadjuk neki a listában megjelenített String-et (ami UGYANAZ a String, mint amit eredetileg a kiszolgálótól kaptunk, amikor meghívtuk a getServiceList()-et). A kiszolgáló visszaadja a tényleges szolgáltatást (sorosítva), aminek a visszafejtésére (az RMI-nek köszönhetően) automatikusan sor kerül, majd egyszerűen meghívjuk a getGuiPanel()-t a szolgáltatásra, és az eredményt (egy JPanel-t) hozzáadjuk a böngésző főpaneléhez (mainPanel).

```
Object[] getServicesList() {
   Object obj = null;
   Object[] services = null;

   try {

      obj = Naming.lookup("rmi://127.0.0.1/ServiceServer");

   }
   catch(Exception ex) {
    ex.printStackTrace();
   }
   server = (ServiceServer) obj;


   try {

     services = server.getServiceList();

   } catch(Exception ex) {
      ex.printStackTrace();
   }
   return services;

}

class MyListListener implements ActionListener {
   public void actionPerformed(ActionEvent ev) {

       Object selection =  serviceList.getSelectedItem();
       loadService(selection);
   }
 }

public static void main(String[] args) {
   new ServiceBrowser().buildGUI();
}
```

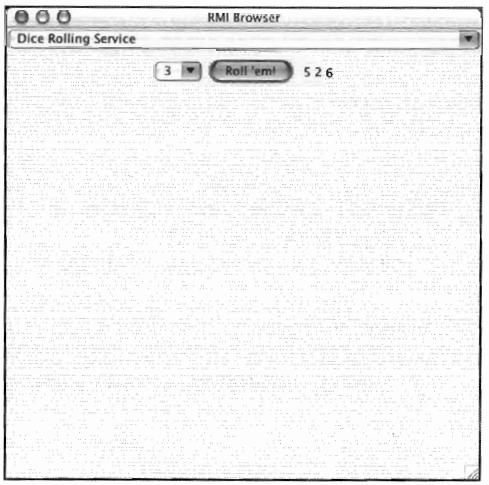
Végrehajtjuk az RMI-keresést, és megszerezzük a csonkot.

A csonkot a távoli szolgáltatás típusára alakítjuk, hogy meghívhassuk rá a getServiceList() tagfüggvényt.

A getServiceList() visszaadja nekünk az Object-ekből álló tömböt, amelyet a felhasználó számára megjelenítünk a JComboBox-ban, hogy választhasson egy szolgáltatást.

Ha ide jutunk, az azt jelenti, hogy a felhasználó választott valamit a JComboBox listájából, ezért fogjuk a választott elemet, és betöltjük a megfelelő szolgáltatást (lásd az előző oldalon szereplő loadService tagfüggvényt, amely elkéri a választásnak megfelelő szolgáltatást a kiszolgálótól).

class DiceService (univerzális szolgáltatás, amely megvalósítja a Service felületet)




```
import javax.swing.*;
import java.awt.event.*;
import java.io.*;

public class DiceService implements Service {

    JLabel label;
    JComboBox numOfDice;



    public JPanel getGuiPanel() {
       JPanel panel = new JPanel();
       JButton button = new JButton("Roll 'em!");
       String[] choices = {"1", "2", "3", "4", "5"};
       numOfDice = new JComboBox(choices);
       label = new JLabel("dice values here");
       button.addActionListener(new RollEmListener());
       panel.add(numOfDice);
       panel.add(button);
       panel.add(label);
       return panel;
    }

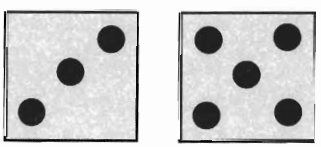
  public class RollEmListener implements ActionListener {
     public void actionPerformed(ActionEvent ev) {
        // dobunk a kockával
        String diceOutput = "";
        String selection = (String)  numOfDice.getSelectedItem();
        int numOfDiceToRoll = Integer.parseInt(selection);
        for (int i = 0; i < numOfDiceToRoll; i++) {
           int r = (int) ((Math.random() * 6) + 1);
           diceOutput += (" " + r);
        }
       label.setText(diceOutput);

     }
    }
```

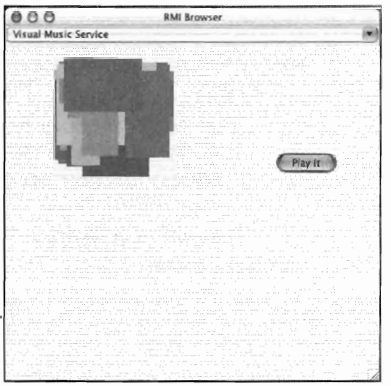
Ez a lényeges tagfüggvény: a Service felület tagfüggvénye, amelyet az ügyfél akkor hív meg, ha ezt a szolgáltatást választották betöltésre. A getGuiPanel() tagfüggvényben tetszőleges dolgot csinálhatsz, feltéve, hogy egy JPanel-t adsz vissza, hogy felépülhessen a tényleges kockadobó grafikus felület.

Hegyezd ki a ceruzád!

Gondolkodj el azon, hogyan javíthatnál a DiceService szolgáltatáson. Egy javaslat: a grafikus felületről tanultak felhasználásával alkoss grafikus dobókockát. Rajzolj ki egy négyszöget annyi körrel, amennyi az adott dobásnak megfelel.



class MiniMusicService (univerzális szolgáltatás, amely megvalósítja a Service felületet)




```
import javax.sound.midi.*;
import java.io.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MiniMusicService implements Service {

    MyDrawPanel myPanel;

    public JPanel getGuiPanel() {
       JPanel mainPanel = new JPanel();
       myPanel = new MyDrawPanel();
       JButton playItButton = new JButton("Play it");
       playItButton.addActionListener(new PlayItListener());
       mainPanel.add(myPanel);
       mainPanel.add(playItButton);
       return mainPanel;
    }

    public class PlayItListener implements ActionListener {
      public void actionPerformed(ActionEvent ev) {

        try {

         Sequencer sequencer = MidiSystem.getSequencer();
         sequencer.open();

         sequencer.addControllerEventListener(myPanel, new int[] {127});
         Sequence seq = new Sequence(Sequence.PPQ, 4);
         Track track = seq.createTrack();

         for (int i = 0; i < 100; i+= 4) {

            int  rNum = (int) ((Math.random() * 50) + 1);
            if (rNum < 38) {  // so now only do it if num <38 (75% of the time)
              track.add(makeEvent(144,1,rNum,100,i));
              track.add(makeEvent(176,1,127,0,i));
              track.add(makeEvent(128,1,rNum,100,i + 2));
            }
         } // end loop

         sequencer.setSequence(seq);
         sequencer.start();
         sequencer.setTempoInBPM(220);
       } catch (Exception ex) {ex.printStackTrace();}

     } // close actionperformed
    } // close inner class
```

A szolgáltatás tagfüggvénye. Összesen annyit tesz, hogy megjelenít egy gombot, valamint a rajzfelületet (ahová majd a négyszögeket festi a program).

Ez ugyanaz a zenelejátszó, mint amit a 12. fejezet Kódkonyhájában láttál, ezért nem látjuk el ismét megjegyzésekkel.

class MiniMusicService – folytatás...

```
public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
     MidiEvent event = null;
        try {
           ShortMessage a = new ShortMessage();
           a.setMessage(comd, chan, one, two);
           event = new MidiEvent(a, tick);

         }catch(Exception e) { }
     return event;
     }


class MyDrawPanel extends JPanel implements ControllerEventListener {

     // only if we got an event do we want to paint
     boolean msg = false;

     public void controlChange(ShortMessage event) {
        msg = true;
        repaint();
     }

     public Dimension getPreferredSize() {
       return new Dimension(300,300);
     }

     public void paintComponent(Graphics g) {
        if (msg) {

          Graphics2D g2 = (Graphics2D) g;

          int r = (int) (Math.random() * 250);
          int gr = (int) (Math.random() * 250);
          int b = (int) (Math.random() * 250);

          g.setColor(new Color(r,gr,b));

          int ht = (int) ((Math.random() * 120) + 10);
          int width = (int) ((Math.random() * 120) + 10);

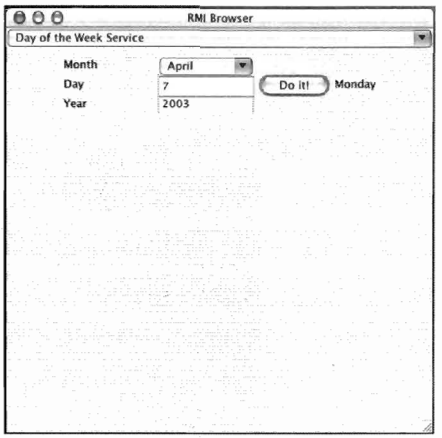
          int x = (int) ((Math.random() * 40) + 10);
          int y =  (int) ((Math.random() * 40) + 10);

          g.fillRect(x,y,ht, width);
          msg = false;

       } // close if
     } // close method
   }  // close inner class
} // close class
```

Ezen az oldalon egyáltalán nincs újdonság: az egészet láttad már a rajzolásról szóló Kódkonyhában. Ha gyakorolni szeretnél, lásd el megjegyzésekkel a kódot, majd hasonlítsd össze az „Egy nagyon képszerű történet" című fejezetben szereplő Kódkonyha kódjával.

class DayOfTheWeekService (univerzális szolgáltatás, amely megvalósítja a Service felületet)




```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.io.*;
import java.util.*;
import java.text.*;

public class DayOfTheWeekService implements Service {

    JLabel outputLabel;
    JComboBox month;
    JTextField day;
    JTextField year;

    public JPanel getGuiPanel() {
       JPanel panel = new JPanel();
       JButton button = new JButton("Do it!");
       button.addActionListener(new DoItListener());
       outputLabel = new JLabel("date appears here");
       DateFormatSymbols dateStuff = new DateFormatSymbols();
       month = new JComboBox(dateStuff.getMonths());
       day = new JTextField(8);
       year = new JTextField(8);
       JPanel inputPanel = new JPanel(new GridLayout(3,2));
       inputPanel.add(new JLabel("Month"));
       inputPanel.add(month);
       inputPanel.add(new JLabel("Day"));
       inputPanel.add(day);
       inputPanel.add(new JLabel("Year"));
       inputPanel.add(year);
       panel.add(inputPanel);
       panel.add(button);
       panel.add(outputLabel);
       return panel;
    }

    public class DoItListener implements ActionListener {
       public void actionPerformed(ActionEvent ev) {
            int monthNum = month.getSelectedIndex();
            int dayNum = Integer.parseInt(day.getText());
            int yearNum = Integer.parseInt(year.getText());
            Calendar c = Calendar.getInstance();
            c.set(Calendar.MONTH, monthNum);
            c.set(Calendar.DAY_OF_MONTH, dayNum);
            c.set(Calendar.YEAR, yearNum);
            Date date = c.getTime();
            String dayOfWeek = (new SimpleDateFormat("EEEE")).format(date);
            outputLabel.setText(dayOfWeek);
       }
    }
}
```

A Service felület tagfüggvénye, amely felépíti a grafikus felületet.

Ha emlékeztetőre van szükséged a szám- és dátumformázás működéséről, lapozd fel ismét a 10. fejezetet. Ez a kód ugyanakkor kissé különbözik az ott látott kódtól, mert a Calendar osztályra támaszkodik. Ezenkívül a SimpleDateFormat lehetővé teszi, hogy megadjunk egy mintát a dátum kiírásához.

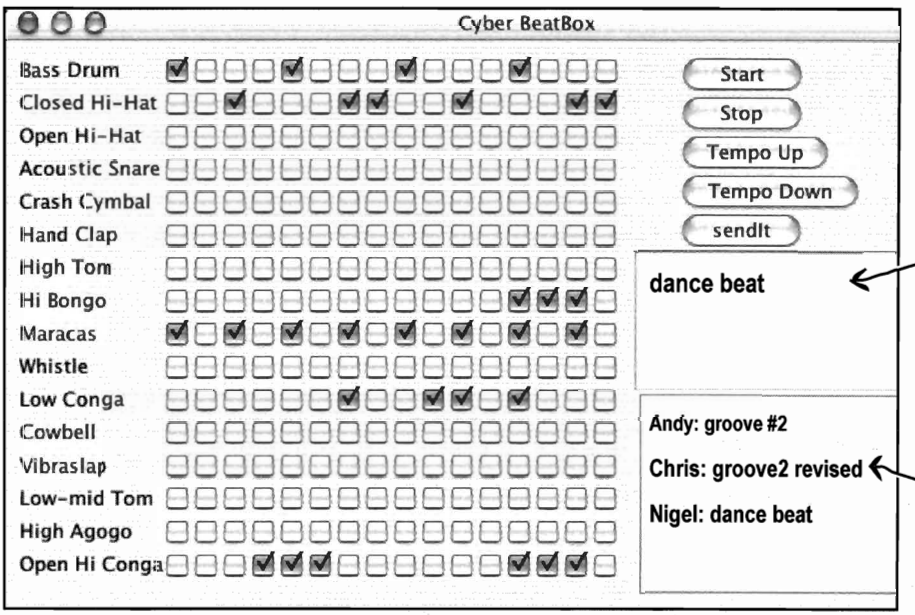


Gratulálunk!

Végigcsináltad!

Persze még hátravan a két függelék.
Meg a tárgymutató.
Aztán ott a webhely...
Hidd el, nincs menekvés.

A függelék
Az Utolsó Kódkonyha



Ide jön az üzenet, amit el szeretnél küldeni a többieknek az adott dobtémával, amikor a sendit („küldd el") gombra kattintasz.

Itt jelennek meg a többiektől érkező üzenetek. Ha rákattintasz valamelyikre, betöltődik az üzenettel kapott téma, és a Start gombbal elindíthatod a lejátszását.

Végre, a BeatBox teljes változata!
A program egy egyszerű MusicServer kiszolgálóhoz kapcsolódik, hogy ritmusmintákat cserélhess a többi felhasználóval.

A végső BeatBox-ügyfélprogram

Ennek a kódnak a nagy része megegyezik a korábbi fejezetek Kódkonyháiban szereplő kódokkal, ezért nem láttuk el ismét megjegyzésekkel az egészet. Az új részek a következők:
GRAFIKUS FELHASZNÁLÓI FELÜLET – Két új összetevőt adtunk a felülethez: a beérkező üzeneteket megjelenítő szövegterületet (valójában gördíthető listát), illetve a szövegmezőt.
HÁLÓZATKEZELÉS – A fejezetben szereplő SimpleChatClient-hez hasonlóan a BeatBox most már kapcsolódik a kiszolgálóhoz, és hozzájut egy bemeneti, illetve kimeneti adatfolyamhoz.
SZÁLAK – Szintén a SimpleChatClient-hez hasonlóan, elindítunk egy „olvasó" (reader) osztályt, amely folyamatosan figyeli a kiszolgálótól érkező üzeneteket. Egyszerű szöveg helyett azonban a bejövő üzenetek KÉT objektumot tartalmaznak: a karakterlánc formátumú üzenetet és a sorosított tömblistát (ami a jelölőnégyzetek állapotát tárolja).

```
import java.awt.*;
import javax.swing.*;
import java.io.*;
import javax.sound.midi.*;
import java.util.*;
import java.awt.event.*;
import java.net.*;
import javax.swing.event.*;

public class BeatBoxFinal {

      JFrame theFrame;
      JPanel mainPanel;
      JList incomingList;
      JTextField userMessage;
      ArrayList<JCheckBox> checkboxList;
      int nextNum;
      Vector<String> listVector = new Vector<String>();
      String userName;
      ObjectOutputStream out;
      ObjectInputStream in;
      HashMap<String, boolean[]> otherSeqsMap = new HashMap<String, boolean[]>();

      Sequencer sequencer;
      Sequence sequence;
      Sequence mySequence = null;
      Track track;


      String[] instrumentNames = {"Bass Drum", "Closed Hi-Hat", "Open Hi-Hat","Acoustic
      Snare", "Crash Cymbal", "Hand Clap", "High Tom", "Hi Bongo", "Maracas", "Whistle",
      "Low Conga", "Cowbell", "Vibraslap", "Low-mid Tom", "High Agogo", "Open Hi Conga"};

      int[] instruments = {35,42,46,38,49,39,50,60,70,72,64,56,58,47,67,63};
```

```
public static void main (String[] args) {
   new BeatBoxFinal().startUp(args[0]);  // args[0] is your user ID/screen name
}

public void startUp(String name) {
    userName = name;
    // open connection to the server
    try {
      Socket sock = new Socket("127.0.0.1", 4242);
      out = new ObjectOutputStream(sock.getOutputStream());
      in = new ObjectInputStream(sock.getInputStream());
      Thread remote = new Thread(new RemoteReader());
      remote.start();
    } catch(Exception ex) {
        System.out.println("couldn't connect - you'll have to play alone.");
    }
    setUpMidi();
    buildGUI();
} // close startUp

 public void buildGUI() {

    theFrame = new JFrame("Cyber BeatBox");
    BorderLayout layout = new BorderLayout();
    JPanel background = new JPanel(layout);
    background.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));

    checkboxList = new ArrayList<JCheckBox>();

    Box buttonBox = new Box(BoxLayout.Y_AXIS);
    JButton start = new JButton("Start");
    start.addActionListener(new MyStartListener());
    buttonBox.add(start);

    JButton stop = new JButton("Stop");
    stop.addActionListener(new MyStopListener());
    buttonBox.add(stop);

    JButton upTempo = new JButton("Tempo Up");
    upTempo.addActionListener(new MyUpTempoListener());
    buttonBox.add(upTempo);

    JButton downTempo = new JButton("Tempo Down");
    downTempo.addActionListener(new MyDownTempoListener());
    buttonBox.add(downTempo);

    JButton sendIt = new JButton("sendIt");
    sendIt.addActionListener(new MySendListener());
    buttonBox.add(sendIt);

    userMessage = new JTextField();
```

Megadunk egy parancssori argumentumot a képernyős neved számára.
Például: %java BeatBoxFinal theFlash

Itt nincs semmi új... Beállítjuk a hálózatkezelést, a bemenetet-kimenetet, és létrehozzuk (illetve elindítjuk) az olvasó szálat.

GUI-kód, semmi újdonság.

```
    buttonBox.add(userMessage);

    incomingList = new JList();
    incomingList.addListSelectionListener(new MyListSelectionListener());
    incomingList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    JScrollPane theList = new JScrollPane(incomingList);
    buttonBox.add(theList);
    incomingList.setListData(listVector); // no data to start with

    Box nameBox = new Box(BoxLayout.Y_AXIS);
    for (int i = 0; i < 16; i++) {
        nameBox.add(new Label(instrumentNames[i]));
    }

    background.add(BorderLayout.EAST, buttonBox);
    background.add(BorderLayout.WEST, nameBox);

    theFrame.getContentPane().add(background);
    GridLayout grid = new GridLayout(16,16);
    grid.setVgap(1);
    grid.setHgap(2);
    mainPanel = new JPanel(grid);
    background.add(BorderLayout.CENTER, mainPanel);

    for (int i = 0; i < 256; i++) {
        JCheckBox c = new JCheckBox();
        c.setSelected(false);
        checkboxList.add(c);
        mainPanel.add(c);
    } // end loop

    theFrame.setBounds(50,50,300,300);
    theFrame.pack();
    theFrame.setVisible(true);
  } // close buildGUI

  public void setUpMidi() {
     try {
     sequencer = MidiSystem.getSequencer();
     sequencer.open();
     sequence = new Sequence(Sequence.PPQ,4);
     track = sequence.createTrack();
     sequencer.setTempoInBPM(120);
  } catch(Exception e) {e.printStackTrace();}

} // close setUpMidi
```

A JList egy olyan összetevő, amelyet még nem használtunk. Ebben jelenítjük meg a beérkező üzeneteket. A szokványos csevegésnél csak MEGNÉZZÜK az üzeneteket, ebben az alkalmazásban azonban ki is VÁLASZTHATSZ egy üzenetet a listából, hogy betöltsd és lejátszd a hozzá csatolt ritmusmintát.

Ezen az oldalon nincs több újdonság.

Megszerezzük a sorozat-feldolgozót (Sequencer), és létrehozunk egy utasítássorozatot (Sequence), illetve egy sávot (Track).

```
public void buildTrackAndStart() {
   ArrayList<Integer> trackList = null; // this will hold the instruments for each
   sequence.deleteTrack(track);
   track = sequence.createTrack();

   for (int i = 0; i < 16; i++) {

     trackList = new ArrayList<Integer>();

     for (int j = 0; j < 16; j++) {
        JCheckBox jc = (JCheckBox) checkboxList.get(j + (16*i));
        if (jc.isSelected()) {
           int key = instruments[i];
           trackList.add(new Integer(key));
        } else {
          trackList.add(null);  // because this slot should be empty in the track
        }
     } // close inner loop
     makeTracks(trackList);
   } // close outer loop
   track.add(makeEvent(192,9,1,0,15)); // - so we always go to full 16 beats
   try {
      sequencer.setSequence(sequence);
      sequencer.setLoopCount(sequencer.LOOP_CONTINUOUSLY);
      sequencer.start();
      sequencer.setTempoInBPM(120);
  } catch(Exception e) {e.printStackTrace();}
 } // close method

public class MyStartListener implements ActionListener {
   public void actionPerformed(ActionEvent a) {
      buildTrackAndStart();
   } // close actionPerformed
} // close inner class

public class MyStopListener implements ActionListener {
   public void actionPerformed(ActionEvent a) {
      sequencer.stop();
  } // close actionPerformed
} // close inner class

public class MyUpTempoListener implements ActionListener {
   public void actionPerformed(ActionEvent a) {
      float tempoFactor = sequencer.getTempoFactor();
      sequencer.setTempoFactor((float)(tempoFactor * 1.03));
  } // close actionPerformed
} // close inner class
```

Végigmegyünk a jelölőnégyzeteken, és az állapotukat kiolvasva, illetve egy hangszerhez rendelve (és a hozzá tartozó MidiEvent-et létrehozva) felépítünk egy sávot. Ez elég bonyolult, de PONTOSAN megegyezik a korábban látottakkal. A teljes magyarázatot megtalálod a korábbi Kódkonyhákban.

A GUI-figyelők. Pontosan ugyanazok, mint a korábbi változatban.

```
    public class MyDownTempoListener implements ActionListener {
        public void actionPerformed(ActionEvent a) {
          float tempoFactor = sequencer.getTempoFactor();
          sequencer.setTempoFactor((float)(tempoFactor * .97));
        }
    }

    public class MySendListener implements ActionListener {
       public void actionPerformed(ActionEvent a) {
           // make an arraylist of just the STATE of the checkboxes
            boolean[] checkboxState = new boolean[256];
           for (int i = 0; i < 256; i++) {
              JCheckBox check = (JCheckBox) checkboxList.get(i);
              if (check.isSelected()) {
                checkboxState[i] = true;
              }
           } // close loop
          String messageToSend = null;
          try {
             out.writeObject(userName + nextNum++ + ": " + userMessage.getText());
             out.writeObject(checkboxState);
          } catch(Exception ex) {
              System.out.println("Sorry dude. Could not send it to the server.");
          }
          userMessage.setText("");
       } // close actionPerformed
    } // close inner class

    public class MyListSelectionListener implements ListSelectionListener {
        public void valueChanged(ListSelectionEvent le) {
          if (!le.getValueIsAdjusting()) {
            String selected = (String) incomingList.getSelectedValue();
            if (selected != null) {
                // now go to the map, and change the sequence
                 boolean[] selectedState = (boolean[]) otherSeqsMap.get(selected);
                 changeSequence(selectedState);
                 sequencer.stop();
                 buildTrackAndStart();
            }
          }
       } // close valueChanged
    } // close inner class
```

Ez új... nagyon hasonlít a SimpleChatClient-re, csak éppen egy String formátumú üzenet küldése helyett két objektumot (az üzenetet és a dobmintát) sorosítunk, és ezt a két objektumot a (kiszolgálóhoz vezető) csatoló kimeneti adatfolyamába írjuk.

Ez szintén új – egy ListSelectionListener, amely értesít, ha a felhasználó kiválasztott egy üzenetet a listából. Amikor a felhasználó kiválaszt egy üzenetet, AZONNAL betöltjük a hozzá társított dobmintát (ami az otherSeqsMap nevű HashMap-ben található), és elindítjuk a lejátszását. Az if tesztekre a ListSelectionEvent események kiolvasásával kapcsolatos kisebb nehézségek miatt van szükség.

```
public class RemoteReader implements Runnable {
    boolean[] checkboxState = null;
    String nameToShow = null;
    Object obj = null;
    public void run() {
      try {
        while((obj=in.readObject()) != null) {
          System.out.println("got an object from server");
          System.out.println(obj.getClass());
          String nameToShow = (String) obj;
          checkboxState = (boolean[]) in.readObject();
          otherSeqsMap.put(nameToShow, checkboxState);
          listVector.add(nameToShow);
          incomingList.setListData(listVector);
        } // close while
      } catch(Exception ex) {ex.printStackTrace();}
    } // close run
} // close inner class
```

Ez a szál feladata: adatokat beolvasni a kiszolgálóról. Ebben a kódban az „adatokat" mindig két sorosított objektum jelenti: a String formátumú üzenet és a dobminta (a jelölőnégyzetek állapotértékeit tartalmazó ArrayList).

Amikor üzenet érkezik, beolvassuk (visszafejtjük) a két objektumot (az üzenetet és a logikai jelölőnégyzet-állapotértékek tömblistáját), és hozzáadjuk a JList összetevőhöz. A JList-hez történő hozzáadás két lépésből áll: létre kell hoznunk egy vektort a listaadatokhoz (a Vector egy régimódi ArrayList), majd utasítanunk kell a JList-et, hogy ezt a vektort használja a listában megjelenítendő adatok forrásaként.

```
public class MyPlayMineListener implements ActionListener {
   public void actionPerformed(ActionEvent a) {
      if (mySequence != null) {
         sequence = mySequence;  // restore to my original
      }
   } // close actionPerformed
} // close inner class
```

Ez a tagfüggvény akkor hívódik meg, amikor a felhasználó kiválaszt valamit a listából. Ekkor AZONNAL a választott mintára váltunk.

```
public void changeSequence(boolean[] checkboxState) {
   for (int i = 0; i < 256; i++) {
      JCheckBox check = (JCheckBox) checkboxList.get(i);
       if (checkboxState[i]) {
           check.setSelected(true);
       } else {
           check.setSelected(false);
       }
   } // close loop
} // close changeSequence
```

A MIDI-kezelő kód teljes egészében megegyezik a korábbi változatban látottal.

```
public void makeTracks(ArrayList list) {
   Iterator it = list.iterator();
   for (int i = 0; i < 16; i++) {
      Integer num = (Integer) it.next();
      if (num != null) {
         int numKey = num.intValue();
         track.add(makeEvent(144,9,numKey, 100, i));
         track.add(makeEvent(128,9,numKey,100, i + 1));
       }
    } // close loop
 } // close makeTracks()
```

```
    public  MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
        MidiEvent event = null;
        try {
          ShortMessage a = new ShortMessage();
          a.setMessage(comd, chan, one, two);
          event = new MidiEvent(a, tick);
        }catch(Exception e) { }
        return event;
    } // close makeEvent
} // close class
```

Semmi újdonság. Pontosan olyan, mint a legutóbbi változat.



Hegyezd ki a ceruzád!

Hogyan javíthatnál még ezen a programon?
Kezdőlökésként íme néhány ötlet:
1) Amikor kiválasztasz egy mintát, az éppen betöltött minta örökre eltűnik. Ha ez egy új minta volt, amelyen éppen dolgoztál (vagy egy korábbi minta módosított változata), nincs szerencséd. Érdemes lehet megjeleníteni egy előugró párbeszédablakot, ami megkérdezi a felhasználót, hogy szeretné-e menteni az aktuális mintát.
2) Ha nem írsz be parancssori argumentumot, futtatáskor csak egy kivételt kapsz. Tegyél valamilyen kódot a main tagfüggvénybe, ami ellenőrzi, hogy a felhasználó átadott-e egy parancssori argumentumot. Ha nem, akkor válassz egy alapértelmezettet, vagy írj ki egy üzenetet, ami arra kéri a felhasználót, hogy futtassa újra a programot, de ezúttal argumentumként megadva a képernyős nevét.
3) Hasznos szolgáltatás lehet, ha egy gombra kattintva arra kérheted a programot, hogy állítson elő egy véletlenszerű mintát. Előfordulhat, hogy valamelyik megtetszik. Ennél is jobb lenne, ha stílusonként (jazz, rock, reggae stb.) betölthetnél egy „alapmintát", amit aztán kiegészíthetsz.
A Head First Java webhelyén kész mintákat találhatsz.

A végső BeatBox-kiszolgálóprogram

Ennek a kódnak a nagy része megegyezik a hálózatkezelésről és a szálakról szóló fejezetben látott SimpleChatServer kódjával. Valójában az egyetlen különbség az, hogy ez a kiszolgáló egy sima karakterlánc (String) helyett két sorosított objektumot fogad és küld tovább (bár a sorosított objektumok egyike történetesen *valóban* String).

```
import java.io.*;
import java.net.*;
import java.util.*;

public class MusicServer {

    ArrayList<ObjectOutputStream> clientOutputStreams;

    public static void main (String[] args) {
         new MusicServer().go();
    }

    public class ClientHandler implements Runnable {

         ObjectInputStream in;
         Socket clientSocket;

         public ClientHandler(Socket socket) {
           try {
             clientSocket = socket;
              in = new ObjectInputStream(clientSocket.getInputStream());

           } catch(Exception ex) {ex.printStackTrace();}
          } // close constructor


        public void run() {
             Object o2 = null;
             Object o1 = null;
           try {

             while ((o1 = in.readObject()) != null) {

                o2 = in.readObject();

                System.out.println("read two objects");
                tellEveryone(o1, o2);
              } // close while

           } catch(Exception ex) {ex.printStackTrace();}
       } // close run
   } // close inner class
```

```
public void go() {
  clientOutputStreams = new ArrayList<ObjectOutputStream>();

     try {
       ServerSocket serverSock = new ServerSocket(4242);

       while(true) {
          Socket clientSocket = serverSock.accept();
          ObjectOutputStream out = new ObjectOutputStream(clientSocket.getOutputStream());
          clientOutputStreams.add(out);

          Thread t = new Thread(new ClientHandler(clientSocket));
          t.start();

          System.out.println("got a connection");
       }
    }catch(Exception ex) {
      ex.printStackTrace();
    }
} // close go

public void tellEveryone(Object one, Object two) {
   Iterator it = clientOutputStreams.iterator();
   while(it.hasNext()) {
     try {
        ObjectOutputStream out = (ObjectOutputStream) it.next();
        out.writeObject(one);
        out.writeObject(two);
      }catch(Exception ex) {ex.printStackTrace();}
    }
} // close tellEveryone

} // close class
```

B függelék
Tíz tuti téma, ami majdnem bekerült a könyvbe...



Nagyon sok mindennel foglalkoztunk, és már majdnem elérkeztél a könyv végéhez. Hiányozni fogsz, de mielőtt elengednénk, nem éreznénk helyesnek, ha nem vérteznénk fel egy kicsit jobban Javaföld viszontagságaival szemben. Mindent, amire szükséged lehet, nem lehet belezsúfolni egy ilyen viszonylag rövid függelékbe. Eredetileg egyébként *megtettük*, és a betűméretet .00003-asra csökkentve idegyömöszöltünk minden olyan Java-ismeretet, amivel a többi fejezetben nem foglalkoztunk. Elfért az egész, csak éppen senki sem tudta elolvasni. Így hát a nagy részét kidobtuk, és csak a tíz legtutibbat tartottuk meg ehhez a függelékhez.

Ez már *tényleg* a könyv vége. A tárgymutatót leszámítva (ami egyébként kötelező olvasmány!).

#10 Bitműveletek

Miért fontos ez?

Beszéltünk róla, hogy egy bájt (byte típus) 8 bitből áll, egy short 16 bitből, és így tovább. Időnként szükség lehet az egyes bitek ki-bekapcsolására. Előfordulhat például, hogy az új, Java-képes kenyérpirítód számára írsz kódot, amikor rájössz, hogy a szűkös memória miatt a kenyérpirító bizonyos beállításai bitszinten szabályozottak. (Az olvashatóság kedvéért a megjegyzésekben csak az utolsó 8 bitet mutatjuk az int-ek teljes 32 bitje helyett.)

Bitenkénti NOT (tagadás) művelet: ~

Ez a művelet „megfordítja" egy primitív összes bitjét.

```
int x = 10; // a bitminta itt 00001010
x = ~x; // a bitminta most 11110101
```

A következő három művelet két primitívet hasonlít össze bitenként, és az összehasonlítás eredményét adja vissza. Az alábbi példákat vesszük alapul:

```
int x = 10; // a bitminta itt 00001010
int y = 6; // a bitminta itt 00000110
```

Bitenkénti AND (és) művelet: &

Ez a művelet egy olyan értéket ad vissza, amelynek csak azok a bitjei lesznek bekapcsolva, amelyek ***mindkét*** eredeti értékben be voltak kapcsolva:

```
int a = x & y; // a bitminta most
               // 00000010
```

Bitenkénti OR (vagy) művelet: |

Ez a művelet egy olyan értéket ad vissza, amelynek csak azok a bitjei lesznek bekapcsolva, amelyek ***vagy az egyik, vagy a másik*** eredeti értékben be voltak kapcsolva:

```
int a = x | y; // a bitminta most
               // 00001110
```

Bitenkénti XOR (kizáró vagy) művelet: ^

Ez a művelet egy olyan értéket ad vissza, amelynek csak azok a bitjei lesznek bekapcsolva, amelyek ***csak az egyik*** eredeti értékben voltak bekapcsolva:

```
int a = x ^ y; // a bitminta most
               // 00001100
```

Eltoló műveletek

Ezek a műveletek egyetlen egész primitívet várnak, és eltolják annak minden bitjét az egyik vagy a másik irányban. Ha leporolod a bináris számtannal kapcsolatos ismereteidet, rájöhetsz, hogy a bitek *balra* tolása lényegében *megszorozza* a számot kettővel, míg a *jobbra* tolás kettővel *osztja* azt.

A következő három művelet bemutatásakor az alábbi példát vessük alapul:

```
int x = -11; // a bitminta itt 11110101
```

Na jó, szólhattunk volna korábban, de itt a világ legrövidebb magyarázata a negatív számok tárolására *kettes komplemens* segítségével. Ne feledd, hogy egy egész szám bal szélső bitje az ***előjelbit***. A Javában egy negatív egész szám előjelbitje *mindig be* van kapcsolva (vagyis 1-es értékre van állítva), míg egy pozitív egész számé mindig *ki*kapcsolt (0). A Java a *kettes komplemens* képlet szerint tárolja a negatív számokat. Egy szám előjele a kettes komplemenssel úgy változtatható meg, hogy megfordítjuk az összes bitjét, majd hozzáadunk 1-et (egy byte esetében ez például azt jelenti, hogy 00000001-et adunk a megfordított értékhez).

Jobbra toló művelet: >>

Ez a művelet egy szám minden bitjét jobbra tolja a megadott számú hellyel, és a bal oldalon álló biteket az eredeti bal szélső bittel tölti fel. Az **előjelbit *nem* változik:**

```
int y = x >> 2; // a bitminta most
                // 11111101
```

Előjel nélküli jobbra toló művelet: >>>

Ugyanazt csinálja, mint a jobbra toló művelet, DE MINDIG nullákkal tölti fel a bal szélső biteket. Az **előjelbit *esetleg* megváltozhat:**

```
int y = x >>> 2; // a bitminta most
                 // 00111101
```

Balra toló művelet: <<

Ugyanazt csinálja, mint az előjel nélküli jobbra toló művelet, de a másik irányban, tehát a jobb szélső biteket tölti fel nullákkal. Az **előjelbit *esetleg* megváltozhat:**

```
int y = x << 2; // a bitminta most
                // 11010100
```

#9 Megváltoztathatatlanság

Miért érdekes, hogy a String-ek megváltoztathatatlanok?

Ha egy Java-program mérete növekedésnek indul, elkerülhetetlen, hogy végül sok-sok String objektumod legyen. Biztonsági okokból – és a memóriával való takarékoskodás érdekében (ne feledd, hogy a Java-programjaid apró Java-képes mobiltelefonokon is képesek futni) – a String-ek (karakterláncok) a Javában megváltoztathatatlanok (immutable). Ez azt jelenti, hogy ha ezt mondod...

```
String s = "0";

for (int x = 1; x < 10; x++) {
  s = s + x;
}
```

...akkor valójában az történik, hogy tíz String objektumot hozol létre ("0", "01", "012"..."0123456789" értékkel). A végén az *s* a "0123456789" értékű String-re fog hivatkozni, de ekkor már *tíz* String létezik!

Amikor új String-et hozol létre, a JVM a memória egy különleges területére helyezi azt, amelyet „karakterláncgyűjtőnek" (String Pool – ugye milyen frissítően hangzik?) neveznek. Ha a karakterláncgyűjtőben már van egy azonos értékű String, akkor a JVM nem hoz létre másolatot belőle, hanem egyszerűen a meglevő elemre mutató hivatkozó változóra hivatkozik.

A JVM ezt azért teheti meg, mert a karakterláncok megváltoztathatatlanok: egy hivatkozó változó nem változtathatja meg egy String értékét egy ugyanarra a String-re mutató másik hivatkozó változóból.

A karakterláncgyűjtővel kapcsolatos másik tudnivaló, hogy a szemétgyűjtő *nem ejti útba.* A példánk *for* ciklusában létrehozott első kilenc String tehát csak csendben vegetál, és pazarolja a memóriát, hacsak később véletlenül létre nem hozol mondjuk egy "01234" értékű karakterláncot.

Hogyan takarít ez meg memóriát?

Nos, ha nem vagy körültekintő, akkor *sehogy*! De ha megérted, hogyan működik a karakterláncok megváltoztathatatlansága, akkor néha kihasználhatod azt, hogy memóriát takaríts meg. Ha azonban sok karakterlánc-műveletet kell végezned (például karakterláncok összefűzését stb.), akkor a StringBuilder nevű, ennek a célnak jobban megfelelő osztályt érdemes használnod.

A StringBuilder-ről hamarosan bővebben is szót ejtünk.

Miért érdekes, hogy a burkolók megváltoztathatatlanok?

A burkolók (wrapper) két fő felhasználási területéről a Math osztályról szóló fejezetben ejtettünk szót:

- Primitív becsomagolása, hogy objektumnak adhassa ki magát
- A statikus segédtagfüggvények (például: Integer.parseInt()) használata

Fontos, hogy megjegyezd, hogy amikor létrehozol egy ilyen burkolóobjektumot...

```
Integer iWrap = new Integer(42);
```

...akkor a burkolóobjektumot nem módosíthatod többé. Az értéke *mindig* 42 lesz. A ***burkolóobjektumoknak nincs beállító tagfüggvénye.*** Azt természetesen megteheted, hogy az *iWrap* egy *másik* burkolóobjektumra mutasson, de akkor *két* objektumod lesz. Miután létrehoztál egy burkolóobjektumot, semmiféleképpen sem változtathatod meg az *értékét*!



#8 Megerősítések

Eddig nem mondtunk sokat arról, hogy miként végezhetsz hibakeresést (debugging) a Java-programjaidban fejlesztés közben. Meggyőződésünk, hogy a Java használatát a parancssorban kell elsajátítanod, ahogy a könyv során is tettük. Ha már profi Java-programozó vagy, dönthetsz úgy, hogy egy IDE-t* használsz, így más hibakereső eszközök is a rendelkezésedre állhatnak. Régen, ha egy Java-programozó hibamentesíteni akarta a kódját, elszórt egy csomó System.out.println() utasítást a programban, kiíratta velük a változók aktuális értékét, illetve néhány „most itt vagyok" üzenetet, hogy lássa, megfelelően halad-e a program folyása. (A 6. fejezet készre sütött kódja benn is hagyott pár hibakereső „print" utasítást a kódban.) Ha a program már helyesen működött, a programozó fogta a System.out.println() utasításokat, és kiszedte azokat a kódból. Ez az eljárás fárasztó és sok hibalehetőséget rejt magában. A Java 1.4 (és 5.0) óta azonban a hibamentesítés lényegesen egyszerűbbé vált. Hogy mi jelentette a megoldást?

A megerősítések

A megerősítések (assertion – feltételezés, állítás) olyanok, mintha szteroidokkal felpumpált System.out.println() utasítások lennének. A kódhoz ugyanúgy adhatod hozzá őket, mint a println utasításokat. A Java 5.0 fordítóprogramja feltételezi, hogy az 5.0-s változatnak megfelelő forrásfájlokat fordíttatsz vele, ezért a Java 5.0 óta a megerősítésekkel történő fordítás alapértelmezés szerint be van kapcsolva. Futásidőben, ha nem csinálsz semmit, a JVM figyelmen kívül hagyja a kódhoz adott assert utasításokat, így azok nem lassítják le a programot. Ha viszont arra utasítod a JVM-et, hogy *kapcsolja be* a megerősítéseket, azok segítenek neked a hibamentesítésben, anélkül, hogy egyetlen kódsort is meg kellene változtatnod. Egyesek panaszkodnak, hogy benne kell hagyniuk az assert utasításaikat a végleges kódban, pedig igazán jól jöhetnek, amikor a kód már a harcmezőn küzd. Ha az ügyfelednek gondja támad, mondhatod neki, hogy futtassa a programot a megerősítések bekapcsolása mellett, és küldje el neked a kimenetet. Ha a megerősítéseket kiszednéd a kódból az üzembe helyezés előtt, erre nem lenne lehetőséged. A megerősítéseknek ráadásul szinte semmilyen hátrányuk nincs: ha nincsenek bekapcsolva, a JVM tökéletesen figyelmen kívül hagyja őket, ezért nem kell sebességcsökkenés miatt aggódnod.

Hogyan működik egy megerősítés?

A kódnak azokon a helyein, ahol úgy véled, hogy valaminek *igaznak kell lennie*, szúrj be megerősítő utasításokat. Például:

```
assert(height > 0);
// ha igaz, a program normál módon folytatódik
// ha hamis, AssertionError-t vált ki
```

A veremkövetéshez kicsit több információt is megadhatsz:

```
assert(height > 0) : "height = "+
height + " weight = " + weight;
```

A kettőspont utáni kifejezés bármilyen érvényes Java-kifejezés lehet, ***amelynek a feloldása nem null értéket eredményez***. Bármit is teszel azonban, soha ***ne hozz létre olyan megerősítéseket, amelyek megváltoztatják egy objektum állapotát***! Ha ugyanis így teszel, akkor a megerősítések futásidejű bekapcsolása megváltoztathatja a program működését.

Fordítás és futtatás megerősítésekkel

Ha egy programot a megerősítésekkel együtt szeretnél *lefordíttatni*, a következő formában teheted meg:

```
javac TestDriveGame.java
```

(Láthatod, hogy semmilyen fordítói kapcsoló nem szükséges.)
Ha megerősítésekkel együtt szeretnéd *futtatni* a programot, ezt a parancsot kell kiadnod:

```
java -ea TestDriveGame
```

*Az IDE az Integrated Development Environment-et (egyesített fejlesztőkörnyezet) jelenti, és olyan eszközöket tartalmaz, mint az Eclipse, a Borland JBuilder vagy a nyílt forrású NetBeans (netbeans.org).

#7 Blokkhatókör

A 9. fejezetben említettük, hogy a helyi változók csak addig élnek, amíg a tagfüggvény, amelyben bevezették őket, a veremben marad. Egyes változóknak azonban még *rövidebb* élettartama lehet. A tagfüggvényeken belül gyakran hozunk létre *kódblokkokat*. A könyvben is gyakran tettünk így, de *kifejezetten* nem beszéltünk a *blokkokról*. A kódblokkok jellemzően tagfüggvényeken belül találhatók, és kapcsos zárójelek – { } – fogják közre őket. Szokványos, jól felismerhető kódblokkok például a ciklusok (*for*, *while*), illetve a feltételes kifejezések (például *if* utasítások).
Nézzünk egy példát:

```
void doStuff() {
    int x = 0;
    for(int y = 0; y < 5; y++) {

        x = x + y;
    }

    x = x * y;
}
```

a tagfüggvény blokkjának kezdete

helyi változó a teljes tagfüggvényre kiterjedő hatókörrel

A for ciklus blokkjának kezdete; az y hatóköre csak a for ciklusra terjed ki!

Nincs gond, az x és az y is hatókörben van.

for ciklus vége

Aú! Ez nem fordítható le! Az y nincs hatókörben! (Más nyelvekben nem biztos, hogy így működik, úgyhogy vigyázat!)

A tagfüggvény blokkjának vége; most már az x is hatókörön kívül van.

A fenti példában az *y* blokkváltozó, amelyet egy blokk belsejében vezettünk be, és amely rögtön kikerül a hatókörből, amint a for ciklus véget ér. A Java-programjaid könnyen hibamentesíthetők és bővíthetők lesznek, ha – amikor csak lehet – példányváltozók helyett helyi változókat használsz, helyi változók helyett pedig blokkváltozókat. A fordító ügyel rá, hogy ne használhass hatókörön kívüli változót, ezért nem kell attól félned, hogy emiatt futás közben hibák lépnek fel.

#6 Láncolt hívás

Ilyesmiből is láttál néhányat ebben a könyvben, bár igyekeztünk a kódokat a lehető legtisztább és legolvashatóbb formába önteni. A Javában ugyanakkor számos engedélyezett rövidítés létezik, amelyekkel bizonyosan találkozni fogsz, különösen ha sok olyan kódot kell olvasnod, amit nem te írtál. Az egyik legelterjedtebb szerkezetet, ami fel fog bukkanni, *láncolt hívásnak* hívják. Nézzünk egy példát:

```
StringBuffer sb = new StringBuffer("spring");
sb = sb.delete(3,6).insert(2,"umme").deleteCharAt(1);
System.out.println("sb = " + sb);
// result is sb = summer
```

Mi a fene történik a második kódsorban? Ez bevallottan mesterkélt példa, de meg kell tanulnod, hogyan értelmezd az ilyesmit.

1. Haladj balról jobbra.
2. Keresd meg a bal szélső tagfüggvényhívás eredményét – ez ebben az esetben az `sb.delete(3,6)`. Ha utánanézel a StringBuffer-nek az API-dokumentációban, láthatod, hogy a `delete()` tagfüggvény egy StringBuffer objektumot ad vissza. A `delete()` tagfüggvény futtatásának eredménye tehát egy StringBuffer objektum lesz, amelynek az értéke `"spr"`.
3. A balról a következő tagfüggvény – `insert()` – az újonnan létrehozott `"spr"` StringBuffer objektumra hívódik meg. Ennek a tagfüggvényhívásnak (az `insert()` tagfüggvénynek) az eredménye *ugyancsak* egy StringBuffer objektum (bár nem muszáj ugyanolyan típusúnak lennie, mint az előző tagfüggvény által visszaadott értéknek). A dolog így folytatódik tovább: a visszaadott objektumot használjuk a jobbra álló következő tagfüggvényhíváshoz. Elméletben tetszőleges számú tagfüggvényt láncolhatsz össze egyetlen utasításban (bár háromnál többet csak ritkán szoktak). Láncolás nélkül a fenti kód második sora olvashatóbb lenne, és valahogy így festene:

```
sb = sb.delete(3,6);
sb = sb.insert(2,"umme");
sb = sb.deleteCharAt(1);
```

De itt egy hétköznapibb és hasznosabb példa, amit már láttál ugyan működés közben, de úgy gondoltuk, hogy nem árt ismét felhívni rá a figyelmet. Arra az esetre való, amikor a main() tagfüggvényednek a főosztály egyik példánytagfüggvényét kell meghívnia, de nem muszáj *hivatkozni* az osztály példányára. Más szavakkal, a main()-nek csak azért kell létrehoznia a példányt, hogy meghívhassa annak *tagfüggvényeit*:

```
class Foo {
   public static void main(String [] args) [
     new Foo().go();
   }
   void go() {
    // here's what we REALLY want...
   }
}
```

Meg akarjuk hívni a go()-t, de a Foo-példány nem érdekes a számunkra, ezért nem vesződünk azzal, hogy az új Foo objektumot egy hivatkozáshoz rendeljük.

#5 Névtelen és statikus beágyazott osztályok

A beágyazott osztályoknak számos fajtája létezik

A könyvnek a grafikus felhasználói felületi események kezelését tárgyaló részében arra használtunk belső (beágyazott) osztályokat, hogy figyelő felületeket valósítsunk meg. A belső osztályoknak ez a legelterjedtebb, legpraktikusabb és leginkább olvasható alkalmazása: az osztályt egyszerűen beágyazzuk egy másik, tartalmazó osztály kapcsos zárójelei közé. Ne feledd, hogy ahhoz, hogy példányt hozhass létre a belső osztályból, szükséged van a külső osztály egy példányára, mert a belső osztály a külső (tartalmazó) osztály *tagja*.
Léteznek ugyanakkor másfajta belső osztályok is, például *statikus* és *névtelen* beágyazott osztályok. A részletekbe itt nem megyünk bele, de nem szeretnénk, ha megdöbbentene a furcsa utasításforma, amikor valaki másnak a kódjában találkozol vele. Lényegében mindent figyelembe véve, amit a Java nyelvvel csinálhatsz, semmi sem eredményez bizarabb megjelenésű kódot, mint a névtelen belső osztályok. De kezdjük valami egyszerűbbel – a statikus beágyazott osztályokkal.

Statikus beágyazott osztályok

Azt már tudod, hogy mit jelent az, hogy statikus: valami olyasmit, ami az osztályhoz, és nem annak egy adott példányához kötődik. A statikus beágyazott osztályok pont úgy néznek ki, mint a nem statikus osztályok, amelyeket az eseményfigyelőkhöz használtunk, azzal a különbséggel, hogy a `static` kulcsszóval vannak megjelölve:

A statikus beágyazott osztályok neve önmagáért beszél: egy olyan osztályról van szó, amelyet egy másik osztály belsejébe ágyaztak be, és amelyet a static módosítóval jelöltek meg.

```
public class FooOuter {
   static class BarInner {
      void sayIt() {
         System.out.println("method of a static inner class");
      }
   }
}

class Test {
   public static void main (String[] args) {
      FooOuter.BarInner foo = new FooOuter.BarInner();
      foo.sayIt();
   }
}
```

Mivel a statikus beágyazott osztály...statikus, nem egy példányát használod a külső osztálynak. Csupán az osztály nevét adod meg, ugyanúgy, ahogy a statikus tagfüggvények meghívásakor vagy a statikus változók elérésekor.

A statikus beágyazott osztályok annyiban jobban hasonlítanak a szokványos, nem beágyazott osztályokra, hogy nincs különleges kapcsolatuk a tartalmazó külső objektummal. Mivel azonban a statikus beágyazott osztályok továbbra is a tartalmazó/külső osztály *tagjának* számítanak, hozzáférhetnek a külső osztály bármely privát tagjához – persze *csak azokhoz, amelyek szintén statikusak*. Mivel a statikus beágyazott osztály nem kapcsolódik a külső osztály egy példányához, nincs rá módja, hogy hozzáférjen a nem statikus (példány-) változókhoz és tagfüggvényekhez.

#5 Névtelen és statikus beágyazott osztályok – folytatás

A beágyazott és a belső közötti különbség

Bármely Java-osztály, amelyet egy másik osztály hatókörén belül határoztak meg, ***beágyazott*** *osztálynak* (nested class) minősül. Az nem érdekes, hogy névtelen, statikus, normál vagy bármilyen más típusú. Ha egy másik osztály belsejében található, akkor technikailag *beágyazott* osztálynak számít. A *nem statikus* beágyazott osztályokra azonban gyakran hivatkoznak *belső* osztályokként (inner class) – mi is így neveztük őket korábban a könyvben. A lényeg az, hogy minden belső osztály beágyazott, de nem minden beágyazott osztály belső.

Névtelen belső osztályok

Képzeld el, hogy valamilyen GUI-kódot írsz, amikor hirtelen rájössz, hogy szükséged van egy olyan osztálypéldányra, amely megvalósítja az ActionListener felületet. Csakhogy *nincs* ActionListener-példányod. Aztán rájössz, hogy nem is írtál *osztályt* ehhez a figyelőhöz. Ekkor két választási lehetőséged van:

1) Létrehozol egy belső osztályt a kódban, ahogy mi is tettük a GUI-kódunkban, majd példányosítod, és átadod ezt a példányt a gomb eseménybejegyző – addActionListener() – tagfüggvényének.

VAGY

2) Létrehozol egy *névtelen* belső osztályt, és ott helyben, futás közben példányosítod. ***Szó szerint ott, ahol vagy, amikor szükséged van a figyelő objektumra.*** Jól értetted: az osztályt és a példányt azon a helyen hozod létre, ahol normális esetben csak a példányt adnád át. Állj meg egy percre, és gondold végig – ez azt jelenti, hogy a teljes *osztályt* adod át ott, ahol szokványos esetben csak egy *példányt* adnál át egy tagfüggvény-argumentumban!

```
import java.awt.event.*;
import javax.swing.*;
public class TestAnon {
   public static void main (String[] args) {

      JFrame frame = new JFrame();
      JButton button = new JButton("click");
      frame.getContentPane().add(button);
      // button.addActionListener(quitListener);

      button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ev) {
          System.exit(0);
        }
      });

   }
}
```

Létrehoztunk egy keretet, és hozzáadtunk egy gombot, most pedig be kell jegyeztetnünk egy eseményfigyelőt a gombhoz. Csakhogy nem írtunk osztályt, ami megvalósítaná az ActionListener felületet...

Normális esetben valami ilyesmit csinálnánk: átadnánk egy hivatkozást egy belső osztály egy példányára...egy olyan belső osztályéra, amely megvalósítja az ActionListener felületet (és az actionPerformed() tagfüggvényt).

Ez az utasítás...

...itt ér véget!

Figyeld meg, hogy azt mondjuk: „new ActionListener()", bár az ActionListener egy felület, így példány NEM HOZHATÓ LÉTRE belőle! Ez az utasításforma valójában azt jelenti, hogy „hozz létre egy új osztályt (név nélkül), amely megvalósítja az ActionListener felületet, ja és mellesleg itt van az actionPerformed() felületi tagfüggvény megvalósítása".

Ahelyett azonban, hogy egy objektumhivatkozást adnánk át...itt az egész új osztály-meghatározást adjuk át!!! Más szavakkal, az ActionListener-t megvalósító osztályt PONTOSAN OTT írjuk meg, AHOL SZÜKSÉGÜNK VAN RÁ. Ez a megoldás automatikusan létrehozza az osztály egy példányát is.

#4 Hozzáférési szintek és elérésmódosítók (ki mit lát?)

A Javában négy *hozzáférési szint* és három *elérésmódosító* létezik. Azért van csak *három* módosító, mert az *alapértelmezés* (amit akkor kapsz, ha nem használsz semmilyen elérésmódosítót) a négy hozzáférési szint egyike.

HOZZÁFÉRÉSI SZINTEK (a legtágabbtól a legszűkebb felé haladó sorrendben)

public ← A public (nyilvános) szint azt jelenti, hogy tetszőleges helyen található tetszőleges kód elérheti a nyilvános elemet („elem" alatt osztályt, változót, tagfüggvényt, konstruktort stb. értünk).

protected ← A protected (védett) szint pontosan úgy működik, mint a default (az azonos csomagban található kódok kapnak hozzáférést), AZZAL A KÜLÖNBSÉGGEL, hogy a csomagon kívüli alosztályoknak is megengedi a védett elem öröklését.

default ← A default (alapértelmezett) hozzáférés azt jelenti, hogy az alapértelmezett hozzáférésű elemet csak az azt tartalmazó osztállyal azonos csomagon belül található kódok érhetik el.

private ← A private (privát) szint azt jelenti, hogy a privát elemet csak a vele azonos osztályban található kódok érhetik el. Ne feledd, hogy a „privát" az osztály számára jelent privátot, nem egy objektum számára. Egy Kutya láthatja egy másik Kutya objektum privát dolgait, de egy Macska nem láthatja egy Kutya privát tagjait.

Elérésmódosítók

```
public
protected
private
```

Legtöbbször csak a public és a private hozzáférési szinteket fogod használni.

public

Nyilvános elérésűvé azokat az osztályokat, állandókat (statikus végleges változókat) és tagfüggvényeket tedd, amelyeket felfedsz más kódok (például beállító- és kiolvasófüggvények) számára, illetve a legtöbb konstruktort.

private

Privát elérésűvé kell tenned lényegében az összes példányváltozót, valamint azokat a tagfüggvényeket, amelyeket nem szeretnél külső kódok számára meghívhatóvá tenni (vagyis azokat a tagfüggvényeket, amelyeket az osztályod nyilvános tagfüggvényei *használnak*).

Mindazonáltal, annak ellenére, hogy a másik két hozzáférési szintet (protected és default) nemigen fogod használni, azt azért tudnod kell, hogy mit csinálnak, mert más kódokban találkozhatsz velük.

#4 Hozzáférési szintek és elérésmódosítók – folytatás

default és protected

default

Mind a védett, mind az alapértelmezett hozzáférési szint csomagokhoz kötődik. Az alapértelmezett hozzáférés egyszerű: annyit tesz, hogy a default hozzáférésű kódot csak a *vele azonos csomagban található* kódok érhetik el. Tehát egy default osztályt (ami olyan osztályt jelent, amelyet nem kifejezetten *public*-ként vezettek be) például csak a default osztállyal azonos csomagon belül levő osztályok érhetnek el.

De mit is jelent valójában *elérni* egy osztályt? Egy olyan kód, amely nem rendelkezik hozzáféréssel egy osztályhoz, még arra sem jogosult, hogy egyáltalán *gondoljon* az osztályra. „Gondolni" alatt itt azt értjük, hogy „használni" az osztályt az említett kódban. Ha a hozzáférés korlátozása miatt például nem érhetsz el egy osztályt, akkor nem is példányosíthatod azt, sőt még az sem megengedett, hogy típusként megadd egy változó, argumentum vagy visszatérési érték számára. Egyszerűen bele sem írhatod a kódba! Ha mégis megteszed, a fordító panaszkodni fog.

Gondold végig, mi következik ebből: ha egy default elérésű osztálynak public tagfüggvényei vannak, akkor azok egyáltalán nem is nyilvánosak, hiszen nem érhetsz el egy tagfüggvényt, ha nem *látod* az osztályt.

De miért akarná bárki egy csomagon belülre korlátozni a hozzáférést? Nos, a csomagok jellemzően olyan osztályokat fognak egy csoportba, amelyek kapcsolódnak egymáshoz, és együtt dolgoznak. Ésszerű tehát, hogy egy adott csomagon belül az osztályok hozzáférhessenek egymás kódjához, miközben együtt, csomagként csak néhány osztályt és tagfüggvényt fednek fel a külvilág (vagyis az említett csomagon kívül található kódok) számára.

Oké, ez volt a default hozzáférés. Nagyon egyszerű: ha valami (egy osztály, változó, tagfüggvény vagy belső osztály) default hozzáférésű (ami, ne feledd, azt jelenti, hogy nincs kifejezett elérésmódosítója!), akkor csak a vele azonos csomagban található kódok érhetik el.

De akkor mire való a *védett* hozzáférés?

protected

A protected (védett) hozzáférés szinte teljesen megegyezik az alapértelmezettel, egyetlen különbségtől eltekintve: a védett elemet az alosztályok *örökölhetik*, még *akkor is, ha ezek az alosztályok az általuk bővített alaposztály csomagján kívül találhatók*. Ennyi az egész. Csupán *ennyit* kínál a protected kulcsszó: a lehetőséget, hogy az alosztályaid függetlenek legyenek az alaposztályuk csomagjától, mégis *örökölhessenek* belőle részeket, például tagfüggvényeket és konstruktorokat.

Sok fejlesztő semmi okot sem lát rá, hogy védett hozzáférést használjon, de azért egyes programokban előfordul, és egyszer-egyszer te is úgy találhatod, hogy éppen erre van szükséged. A védett hozzáférés egyik érdekes vonása, hogy a többi hozzáférési szinttől eltérően csak az *öröklésre* vonatkozik. Ha a csomagon kívüli alosztály *hivatkozik* az alaposztály egy példányára, és az alaposztálynak van mondjuk egy védett tagfüggvénye, az alosztály ezen a hivatkozáson keresztül nem érheti el az említett védett tagfüggvényt! A tagfüggvény elérésnek egyetlen módja, ha az alosztály *örökli* azt. Más szavakkal, a csomagon kívüli alosztály nem rendelkezik hozzáféréssel a védett tagfüggvényhez, csupán *tartalmazza*, mert örökölte.

#3 A String és a StringBuffer/StringBuilder tagfüggvényei

A String és a StringBuffer a Java API leggyakrabban használt osztályai közé tartoznak. (A pár oldallal korábbi #9-es pontból emlékezhetsz rá, hogy a String-ek megváltoztathatatlanok, ezért egy StringBuffer vagy StringBuilder lényegesen hatékonyabb lehet, ha karakterláncokon kell műveleteket végezned.) A Java 5.0 óta a String***Builder*** osztályt kell használnod a String***Buffer*** helyett, hacsak a karakterlánc-műveleteknek szálbiztosaknak nem kell lenniük, ami nem túl gyakori. Íme egy rövid áttekintés az említett osztályok **legfontosabb** tagfüggvényeiről:

Ezekkel a tagfüggvényekkel a String és a StringBuffer/StringBuilder osztály egyaránt rendelkezik:

char charAt(int *index*); // milyen karakter található a megadott pozíción
int length(); // milyen hosszú a karakterlánc
String substring(int *kezdet*, int *vég*); // a karakterlánc egy részének kinyerése
String toString(); // a karakterlánc String értéke

Karakterláncok összefűzéséhez:

String concat(*karakterlánc*); // a String osztályban
String append(*karakterlánc*); // a StringBuffer és StringBuilder osztályokban

Ezek a String osztály tagfüggvényei:

String replace(char *régi*, char *új*); // egy karakter összes előfordulásának lecserélése
String substring(int *kezdet*, int *vég*; // karakterlánc egy részének kinyerése
char[] toCharArray(); // átalakítás karakterek tömbjévé
String toLowerCase(); // az összes karakter kisbetűsre cserélése
String toUpperCase(); // az összes karakter nagybetűsre cserélése
String trim(); // az üreshelyek levágása a karakterlánc végeiről
String valueOf(char[]) // karakterlánc létrehozása egy karaktertömbből
String valueOf(int *i*) // karakterlánc létrehozása egy primitívből
// más primitívek is támogatottak

Ezek pedig a StringBuffer és a StringBuilder osztály tagfüggvényei:

StringBxxxx delete(int *kezdet*, int *vég*); // karakterláncrész törlése
StringBxxxx insert(int *eltolás*, *tetszőleges primitív vagy char[]*); // beszúrás
StringBxxxx replace(int *kezdet*, int *vég*, String *s*); // az adott rész cseréje a megadott karakterláncra
StringBxxxx reverse(); // a StringBuffer/StringBuilder megfordítása
void setCharAt(int *index*, char *karakter*); // a megadott karakter cseréje

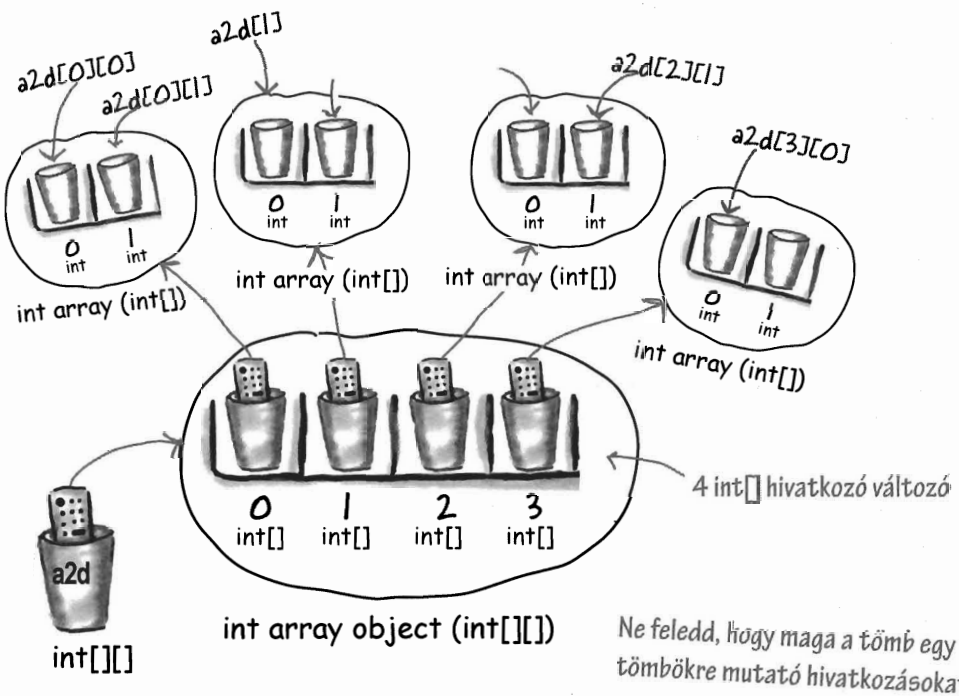
Megjegyzés: a StringB***xxxx*** jelentése a helyzettől függően String*Buffer* vagy String*Builder*.

#2 Többdimenziós tömbök

A legtöbb nyelvben érvényes, hogy ha létrehozol mondjuk egy 4x2-es kétdimenziós tömböt, akkor egy 4x2, vagyis összesen 8 elemet tartalmazó négyszöget kell elképzelned. A Javában azonban egy ilyen tömb valójában 5 összekapcsolt tömböt jelent! A Javában a kétdimenziós tömbök egyszerűen *tömbök tömbjei.* (Egy háromdimenziós tömb pedig tömbök tömbjeinek tömbje, de a sor folytatását meghagyjuk neked.) A dolog így működik:

```
int[][] a2d = new int [4][2];
```

A JVM egy 4 elemű tömböt hoz létre, amelyben valójában a négy elem *mindegyike* egy-egy hivatkozó változó, ami egy (újonnan létrehozott) 2 elemű int tömbre mutat.



A többdimenziós tömbök használata

- A harmadik tömb második elemének elérése: `int x = a2d[2][1];  //ne feledd, 0-tól indul!`
- Egydimenziós hivatkozás létrehozása az egyik altömbre: `int[] copy = a2d[];`
- 2x3-as tömb rövidített előkészítése: `int[][] x = {{2,3,4}, {7,8,9}};`
- Kétdimenziós tömb létrehozása szabálytalan dimenziókkal:

```
int[][] y = new int [2][];  // csak az első tömböt hozza létre, 2 hosszúsággal
y[0] = new int [3];  // az első altömböt 3 elem hosszúvá teszi
y[1] = new int [5];  // a második altömböt 5 elem hosszúvá teszi
```

És a legtutibb téma, amelynek majdnem sikerült bekerülnie...

#1 Felsorolások (felsoroló típusok vagy Enum-ok)

A könyvben szót ejtettünk az API-ban meghatározott állandókról, például a **`JFrame.EXIT_ON_CLOSE`**-ról. Azt is megtanultad, hogy saját állandót is létrehozhatsz, ha egy változót **`static final`**-ként jelölsz meg. Néha azonban előfordulhat, hogy olyan állandó értékekből álló halmazt szeretnél létrehozni, amelyek egy változó ***összes*** érvényes értékét jelképezik. Az ilyen érvényes értékekből álló halmazokat hívják közönségesen *felsorolásnak* (enumeration). A Java 5.0 előtt csak félig-meddig lehetett felsorolást létrehozni a Javában, a Java 5.0 óta azonban teljes értékű felsorolásokat hozhatsz létre, amelyeket a korábbi Java-változatokat használó barátaid irigyelni fognak.

Ki van a bandában?

Tegyük fel, hogy a kedvenc zenekarod számára készítesz webhelyet, és gondoskodni szeretnél róla, hogy minden hozzászólás a banda egy adott tagjához kapcsolódjon.

Az „enum"-ok utánzásának régi módja:

```
public static final int JERRY = 1;
public static final int BOBBY = 2;
public static final int PHIL = 3;

// later in the code

if (selectedBandMember == JERRY) {
  // do JERRY related stuff
```

Reméljük, hogy mire ide jutunk, a „selectedBandMember"-nek („kiválasztott BandaTag") érvényes értéke van!

A jó hír ezzel a megoldással kapcsolatban az, hogy könnyebben olvashatóvá teszi a kódot. A másik jó hír, hogy az általad létrehozott hamis felsorolások értékét soha nem változtathatod meg: `JERRY` mindig `1` lesz. Rossz hír viszont, hogy nincs egyszerű vagy hatékony módja annak, hogy biztosítsd, hogy a `selectedBandMember` értéke mindig `1`, `2` vagy `3` legyen. Ha egy szokatlan kód `812`-re állítja a `selectedBandMember` értékét, nagy a valószínűsége, hogy a kódod működésképtelenné válik...

#1 Felsorolások – folytatás

Lássuk ugyanezt egy valódi Java 5.0-s enum használatával! A felsorolásunk nagyon egyszerű, de a legtöbb felsorolás *tényleg* ilyen.

Az új, hivatalos „enum":

```
public enum Members { JERRY, BOBBY, PHIL };
public Members selectedBandMember;

// later in the code

if (selectedBandMember == Members.JERRY) {
  // do JERRY related stuff
}
```

Ez nagyjából úgy néz ki, mint egy egyszerű osztálymeghatározás, nem? Nos, az enum-ok VALÓBAN különleges osztályok. Itt egy „Members" (tagok) nevű új felsoroló típust hoztunk létre.

A „selectedBandMember" változó „Members" típusú, és KIZÁRÓLAG a „JERRY", „BOBBY" vagy „PHIL" értéket veheti fel.

Egy enum „példányra" ilyen formában lehet hivatkozni.

Ennek a változónak az értéke miatt nem kell aggódnod!

A felsorolás a java.lang.Enum-ot bővíti

Amikor meghatározol egy felsorolást, akkor egy új osztályt hozol létre, és közvetetten a ***java.lang.Enum*** osztályt bővíted. Egy felsorolást bevezethetsz önálló osztályként, saját forrásfájlban, de egy másik osztály tagjaként is.

Az „if" és a „switch" használata Enum-okkal

Az imént létrehozott felsorolást felhasználva elágaztathatod a kódot egy `if` vagy egy `switch` utasítással. Azt is figyeld meg, hogy az enum-példányokat az `==` segítségével vagy az `.equals()` tagfüggvénnyel hasonlíthatod össze. Általában az előbbit tartják jobb kódolási stílusnak.

```
Members n = Members.BOBBY;
if (n.equals(Members.JERRY)) System.out.println("Jerrrry!");
if (n == Members.BOBBY) System.out.println("Rat Dog");

Members ifName = Members.PHIL;
switch (ifName) {
  case JERRY: System.out.print("make it sing ");
  case PHIL: System.out.print("go deep ");
  case BOBBY: System.out.println("Cassidy! ");
}
```

Felsoroló érték hozzárendelése egy változóhoz.

Ez a kettő egyaránt jól működik.
A program a „Rat Dog" szöveget írja ki.

Pop-kvíz! Mi lesz a kimenet?

Válasz: ¡Apissed deep ob

#1 Felsorolások – befejezés

Ugyanaz a felsorolás, rendesen kipofozva

A felsoroláshoz mindenféle dolgot hozzáadhatsz: konstruktort, tagfüggvényeket, változókat, vagy egy olyan dolgot, amit állandófüggő osztálytörzsnek (constant-specific class body) hívnak. Ez nem túl gyakori, de összefuthatsz vele:

```
public class HfjEnum {

  enum Names {
    JERRY("lead guitar") { public String sings() {
                           return "plaintively"; }
                         },
    BOBBY("rhythm guitar") { public String sings() {
                             return "hoarsely"; }
                           },
    PHIL("bass");

    private String instrument;

    Names(String instrument) {
      this.instrument = instrument;
    }
    public String getInstrument() {
      return this.instrument;
    }
    public String sings() {
      return "occasionally";
    }
  }

  public static void main(String[] args) {
    for (Names n : Names.values()) {
      System.out.print(n);
      System.out.print(", instrument: "+ n.getInstrument());
      System.out.println(", sings: " + n.sings());
    }
  }
}
```

Ez egy argumentum, amelyet az alább bevezetett konstruktor kap meg.

Ezek az úgynevezett „állandófüggő osztálytörzsek". Gondolj rájuk úgy, mintha felülbírálnád a felsorolás alapfüggvényét (esetünkben a „sing()" tagfüggvényt), ha azt egy JERRY vagy BOBBY felsorolt értékű változóra hívják meg.

Ez a felsorolás konstruktora, amely minden bevezetett felsorolt értékre egyszer fut le (esetünkben tehát háromszor).

Ezeket a tagfüggvényeket a „main()" hívja meg.

Minden felsorolásnak van egy beépített „values()" tagfüggvénye, amelyet általában egy ehhez hasonló „for" ciklusban használnak.

```
File Edit Window Help Bootleg
%java HfjEnum

JERRY, instrument: lead guitar, sings: plaintively
BOBBY, instrument: rhythm guitar, sings: hoarsely
PHIL, instrument: bass, sings: occasionally
%
```

Vedd észre, hogy a „sing()" alapfüggvény csak akkor hívódik meg, ha a felsorolt értéknek nincs állandófüggő osztálytörzse.



Ötperces rejtély



Hosszú út hazafelé

Byte kapitány, a síkföldi *Utazó* csillaghajó parancsnoka, sürgős, „szigorúan titkos" jelzésű üzenetet kapott a főhadiszállásról. Az üzenet 30, alaposan titkosított navigációs kódot tartalmazott, amelyekre az *Utazónak* szüksége volt ahhoz, hogy sikeresen hazajusson az ellenséges vonalakon keresztül. A szomszédos galaxisból származó hekkeriaiak, akikkel hadban álltak, kifejlesztettek egy ördögi kód-összezavaró sugarat, ami képes volt hamis objektumokat létrehozni az *Utazó* egyetlen navigációs számítógépének kupacában. Ezen felül az idegen sugár az érvényes hivatkozó változókat is módosítani tudta, hogy azok a hamis objektumokra mutassanak. Az *Utazó* legénysége csak egy helyi, a hajó korszerű Java 1.4-es kódjába ágyazható víruskereső futtatásával tudott védekezni a gonosz hekkeriai sugár ellen.
Byte kapitány a következő programozási utasításokat adta Smith zászlósnak a létfontosságú navigációs kódok megfejtéséhez:

„Tegye az első 5 kódot egy ParszekKulcs típusú tömbbe, az utolsó 25-öt pedig egy ötször ötös, KvadransKulcs típusú kétdimenziós tömbbe. Adja át ezt a két tömböt a nyilvános, végleges HajoNavigacio osztály tervezUtvonal() tagfüggvényének. Miután megkapta tőle az útvonalobjektumot, futtassa le a belső víruskeresőt a program minden hivatkozó változójára, majd indítsa el a NavSim programot, és közölje velem az eredményt."

Néhány perccel később Smith zászlós visszatért a NavSim kimenetével. „Uram, a NavSim kimenete készen áll." – jelentette be. „Nagyszerű" – felelte a kapitány – „Ismertesse az elvégzett műveleteket!". „Igenis, uram!" – válaszolta a zászlós – „Először bevezettem és létrehoztam egy ParszekKulcs típusú tömböt ezzel a kóddal: `ParszekKulcs[] p = new ParszekKulcs[5];`, majd bevezettem és létrehoztam egy KvadransKulcs típusú tömböt ezzel a kóddal: `KvadransKulcs[][] q = new KvadransKulcs[5][5];`. Ezt követően betöltöttem az első 5 kódot a ParszekKulcs tömbbe egy 'for' ciklussal, majd az utolsó 25 kódot a KvadransKulcs tömbbe beágyazott 'for' ciklusok segítségével. Ez után lefuttattam a víruskeresőt mind a 32 hivatkozó változóra: egyszer a ParszekKulcs tömbre, ötször a ParszekKulcs elemeire, egyszer a KvadransKulcs tömbre, és huszonötször a KvadransKulcs elemeire. Miután a víruskereső nem észlelt vírust, lefuttattam a NavSim programot, aztán a biztonság kedvéért újra lefuttattam a víruskeresőt, uram."
Byte kapitány egy percig fagyos arccal a zászlósra meredt, majd nyugodt hangon megszólalt: „Zászlós, a hajó biztonságának veszélyeztetése miatt további parancsig nem hagyhatja el a körletét. Nem akarom látni a hídon addig, amíg tiszteségessen meg nem tanulta a Java használatát! Boolean hadnagy, vegye át a zászlós helyét, és hajtsa végre a feladatot rendesen!"

Miért ítélte szobafogságra a kapitány a zászlóst?



Ötperces rejtély – megoldás



Hosszú út hazafelé

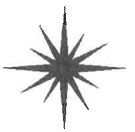
Byte kapitány tudta, hogy a Javában egy többdimenziós tömb valójában tömbök tömbje. Az ötször ötös "q" KvadransKulcs tömbnek tehát összesen 31 hivatkozó változóra volt szüksége ahhoz, hogy minden eleméhez hozzáférjen:

1 – a "q" hivatkozó változója
5 – a q[0]–q[4] elemek hivatkozó változói
25 – a q[0][0]–q[4][4] tömbelemek hivatkozó változói

A zászlós megfeledkezett a "q" tömbbe beágyazott 5 egydimenziós tömb hivatkozó változóiról, márpedig ezek bármelyikét megfertőzhette volna a hekkeriai sugár, és a zászlós tesztje soha nem fedte volna fel a problémát.




Tárgymutató



A, Á

abstract 200
accessor 79
Adatszerkezetek 529
add(Object elem) 132
aktív szál 491
aktuális objektum 256
alapkönyvtár 59
alaposztály 166
alaptípus 49
Alaptípusok átalakítása 117
alkalmazás-programozási felület 125
alkalmazkodó felderítés 633
álkód 99
állandó 41
állandófüggő osztálytörzs 673
állandók 282
állapot 34, 71, 429
alosztály 166
általános tagfüggvények 544
általános típusok 540, 568
általánosítás 539, 570
animáció 382
API-csomagok 154
API-dokumentáció 160
argumentum 74
argumentum nélküli konstruktor 245
ArrayList 132-133, 155
ArrayList API 532
assertion 662
átmeneti tár 453
atomi művelet 510

B

beállító 79
befőttesüveg 585
behavior 71, 429
belső osztály 376
Betokozás 80-81
betöltés 157
BindException 477
bitműveletek 660
blokkváltozó 663
BorderLayout 403
BoxLayout 403
break 105
buffer 453
BufferedReader 478

burkoló tagfüggvények 292
burkolók 661
burkolóosztályok 287
byte 51

C, Cs

Calendar 304-305
Calendar API 306
cast 117
chaining 432
cikluskifejezés 114
Ciklusok 10-11, 115
ciklusváltozó 116
class 7
ClassCastException 216
classes könyvtár 584
Collections.sort() 534
Comparable 547
compare() 552
compareTo() 549
contains(Object elem) 132
ControllerEvent 387
csatolók 471
csatolókapcsolat létesítése 475
csatolókapcsolatok 474
csevegőprogram 473
csomag 587, 589
csomagnevek 588
csomagok 154
csonk 103, 614, 618
csontváz 614, 618
csökkentő művelet 109
csúcsos zárójel 539
csúcsos zárójelek 137

D

–d kapcsoló 591
dátumok 301
deadlock 514
debugging 662
decrement 109
deserialization 441
dinamikus osztályletöltés 621
dobozelrendezés 403, 411
dobozolás 289
double 51
ducking 325

E, É

EJB 607, 631
Elágaztatás 10
elaltatás 501
elérésmódosító 81, 667
elérő 79
életciklus 254
előkészítés 114
előnövelés 115
elrendezéskezelő 399, 401
elvont osztály 197, 201
Elvont tagfüggvények 203
elvont típus 200
Enterprise Java Beans 607
Enterprise JavaBeans 631
Enum 671
equals() 86, 561
érték szerinti átadás 77
értékadás 52
eseményfigyelő 358
eseménykezelés 357, 362
eseményobjektum 358
Exception 322
expression 115
extrém programozás 101

F

fahalmaz 565
fájlbemeneti adatfolyam 441
fájlkimeneti adatfolyam 432
fájlok írása 452
fájlok olvasása 454
felfedés 80
Feltételes elágazások 13
feltételvizsgálat 114
felülbírálás 32, 167
felületek 197, 224
felülírás 32, 167

fenntartott szavak 53
figyelő 359
FileInputStream 441
FileOutputStream 432
final 41, 56
finally 327
float 51
FlowLayout 403
foglalt szó 53
folyamat 489
folyó elrendezés 403, 408
for 105-106
for ciklus 114
for each 116
for in 116
Fordító 2, 19
Formatter 294
formátumleíró 298
formázás 295
formázási argumentumok 300
formázó karakterlánc 297
Forrás 2
forráskód 584
frame 237
Framework 529
Frázisautomata 17, 630
fully-qualified name 587
Futtatás 9

G, Gy

garbage collectible heap 57
generics 539, 548
get(int index) 132
getInstance() 304
getter 79
getUserInput() 111
grafikus felület 353
gyűjtemények 558

H

halmaz 557
hálózati verem 475
hálózatkezelés 471
HAS-A 177
hashCode() 561
HashMap 533
HashSet 533, 559, 561
hatókör 258
heap 236
helper 612
helyettes 612
helyettesítő karakter 574
helyi változó 49, 85, 258, 264
hibamentesítés 662
hívásverem 237
hivatkozás 49, 54, 56
hivatkozó változó 54
holtpont 514
HttpServlet 626

I

immutable 661
implementál 99
import 155
include 157
increment 109
index 61
indexOf(Object elem) 132
inheritance 165
InputStreamReader 478
instance 34
instance variable 34
instanceof 216
int 51
Integer 117
Integer.parseInt() 106
interface 197, 224
InterruptedException 501
IS-A 177
isEmpty() 132
ismétlődés 556, 561
iteration 114

J

J2EE 631
JAR 585

jar fájl 41
–jar kapcsoló 586
Java 1.02 4
Java 1.1 4
Java 2 (1.2–1.4 változatok) 4
Java 2 Enterprise Edition 631
Java 5.0 (1.5 és újabb változatok) 4
Java API 125
Java története 4
Java Web Start 597
java.io.File 452
java.lang 154
java.lang.Enum 672
java.lang.String 157
java.lang.System 157
java.net 471
java.rmi.Naming 617
java.util 154
java.util.Calendar 303
java.util.Date 303
java.util.GregorianCalendar-példány 304
Java-archívum 41, 585
Java-gyűjtemények keretrendszere 529
Java-keret 354
Java-kódok felépítése 7
Java-könyvtár 125
JavaSound 317, 340
javax.swing 154
javax.swing.JComponent 400
JCheckBox 416
jegyzékfájl 585
jelölőnégyzet 416
JFrame 354
Jini 607, 632
jnlp fájl 599
JPanel 364
JScrollPane 414
JSP 628
JTextArea 414
JTextField 413
JVM 18
JWS 597

K

kapu 476
Karakterlánc kiírása 447
karakterláncok 661
keresőszolgáltatás 632
keret 237
keretes elrendezés 403-404
kettőspont 105
kezdőérték 84
kifejezés 115
Kimenet (kód) 2
kiolvasó 79
kisebb tagfüggvények 144
kiszolgáló készítése 483
kiszolgálói csatoló 471
kiszolgálói kisalkalmazás 625
kivételek 320, 322
kódblokk 663
kódvázlat 99
konkrét osztály 200, 202
konstruktor 235, 241
közzététel 581
kulcs szerinti keresés 557
kulcs–érték párok 557
kulcsszó 53
Kupac 40, 58, 236

L

láncolás 432
láncolt hívás 664
láncolt lista 534
Layout Manager 399
lebukás 325, 337
length 62
LinkedHashMap 533
LinkedList 533-534
List 557
literális értékek 52
localhost 478
lock 509
logikai kifejezés 151
logikai vizsgálat 114
long 51
loop 106

M

main 38
main() tagfüggvény 9
manifest 41
Map 557, 567
matematikai tagfüggvények 274, 286
Math 274, 286
Math.random 111
megvalósít 99
Mentés 430
META-INF 593
method 7, 34
Midi-esemény 344
MidiEvent 343
MIDI-üzenet 344
MIDI-zenelejátszó 316
módosító 79
MouseEvent 358
multithreaded 471
mutató 54
mutator 79
műveleti jel 115

N, Ny

naptár 304
nem atomi művelet 512
névütközés 587
normál tömbök 136
növelő művelet 109
null 56
NumberFormatException 292

O, Ö

Object 210
Object osztály 208
ObjectInputStream 441
ObjectOutputStream 432
objektum 35
objektumazonosság 560
objektumbemeneti adatfolyam 441
objektumegyenlőség 86
objektumépítés 252
objektumhivatkozás 50, 54
objektumkimeneti adatfolyam 432
objektumok 27
objektumok átalakítása 216
objektumok létrehozása 36, 240
Objektumok mentése 429, 432
objektumok visszaállítása 441
operátor 115
osztály 7, 35
osztályhierarchia 174
Osztályok 27
osztályok felépítése 8
override 32, 167
Öngyógyító hálózat 635
öröklés 165, 168
Öröklési fa 176
ősosztály 166
összehangolás 512, 514
összehangolt tagfüggvények 511
összehasonlítás 86
összehasonlító 552
összetevő 400

P

package 154
package utasítás 589
paintComponent() 364
paraméter 74, 79
pehelysúlyú objektum 210
példány 34
példányváltozó 34, 72, 85, 264
Példányváltozók bevezetése 84
példányváltozók előkészítése 84
polimorfizmus 165
polymorphism 165
pont művelet 36
port 476
pozíció 61
primitív 49
PrintWriter 479
private 81
process 489
programtervezési minták 181

protokoll 183
proxy 612
pszeudokód 99
public 41, 81

R

rajzolás 363
rajztábla 364
random() 111
Remote Method Invocation 607
remove(int index) 132
remove(Object elem) 132
rendezés 529
rendszerezés 584
return 75
RMI 607, 610
rmic 618
RMI-jegyzék 615
rmiregistry 615, 618
RMI-segédobjektumok 614
Runnable 493

S, Sz

Sáv 340
Save 430
segédek 612
segédosztály 112, 152
Sequencer 318
Serializable 437
serialization 432
serialver 461
serialVersionUID 461
ServerSocket 483
servlet 607, 625
Set 557
setter 79
short 51
Singleton 526
size() 132
skeleton 614
socket 471
sorosítás 432, 438
sorosítás visszafejtése 441
sorosított objektumok 434
sort() 532
source könyvtár 584
split() 458
stack 236
standard library 59
Starbucks 51
state 71, 429
static 41
statikus betöltés 307
statikus tagfüggvények 275, 278
statikus változók 279-280
String 157
String int-té alakítása 105
StringBuilder 661
stub 614
subclass 166
superclass 166
Swing 399
Swing-összetevők 413
synchronized 514
System 157
szál 489
szál indítása 492
szálállapotok 496
szálprioritás 522
szálütemező 496
számformázás 294
számláló 114
százalékjel 296
szemétgyűjtés 40
szemétgyűjtő kupac 54, 57, 236
szolgáltatásböngésző 636
szövegfájl 447
szuperosztály 166

T, Ty

tagfüggvény 7, 34
Tagfüggvény túlterhelése 191
tagfüggvények 34
tagfüggvény-paraméter 74
társalgás 38
tartománynevek 588
távoli felület 616
távoli szolgáltatás 611, 615
távoli tagfüggvényhívás 607, 610
TCP-kapu 476

teljesen minősített név 587
térkép 567
tervezési minták 181
tesztelő 36
tesztkód 99
tesztosztály 36
Thread 489
thread priority 522
Throwable 322
throws 320, 324
típus 50
típusátalakítás 78, 117, 216
típusbiztos gyűjtemények 540
típuskényszerítés 117
típusparaméter 543
torpedójáték 96
toString() 536
továbbfejlesztett for ciklus 106, 116
többalakú tömbök 186
többalakúság 165, 184, 206, 215, 570
többsoros szövegmező 414
többszálú 471
Többszálúság 489
többszörös öröklés 222
tömb 59, 135
tömblista 132
tömbobjektum 59
tömbpozíció 130
tömbutasítás 136
tömbváltozó 59
track 340
transient 439
TreeSet 533-534, 564
try/catch 321, 324, 326
túlcsordulás 52
túlterhelt konstruktor 248

U, Ü

utasítások 7, 10
utónövelés 115
utónövelő művelet 105
ügyfélcsatoló 471
üres hivatkozás 58, 235

V

vállalati szintű Java-babszem 607
valódi kód 99
változatazonosító 461
változatkövetés 460
változó bevezetése 50
változók 49, 236
végrehajtási szál 489
végrehajtható JAR 586
véletlenszám 111
verem 236-237
vezérlőesemény 387
Virtuális gépek 2
viselkedés 34, 71, 429
visszatérési típus 75, 79
void 75

W

while 115
widget 354
wildcard 574
wrapper 661

X

XP 101

Z, Zs

zár 509
zenegép 316

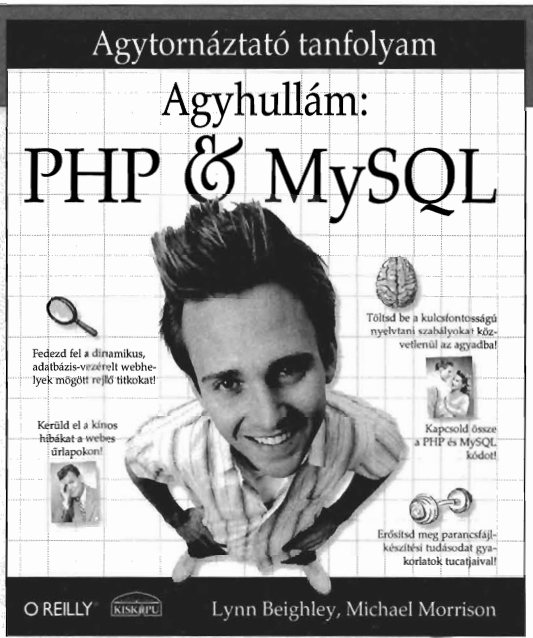
Kiskapu Kiadó ajánlata

Agyhullám: PHP & MySQL

Michael Morrison, Lynn Beighley

Miért néznek ki ezek a könyvek másképpen, mint a többi?

Úgy gondoljuk, hogy az Olvasó ideje túl drága ahhoz, hogy új fogalmak bemagolására pazarolja, ezért a kognitív tudományok és a tanuláselmélet legújabb kutatási eredményeinek felhasználásával egy minden érzékszervre ható anyagot állítottak össze a szerzők. Az *Agyhullám* sorozat képileg is izgalmas formátumát úgy tervezték meg, hogy az álmosító, szöveggel telezsúfolt könyvekkel szemben az agy működéséhez igazodjon.

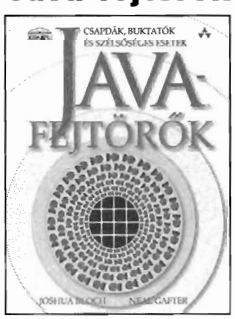


Bolti ár: 12000 Ft
Akciós ár
a Kiskapu webboltban: 9000 Ft

A Kiskapu Kiadó további ajánlata programozók számára:

Joshua Bloch, Neal Gafter:

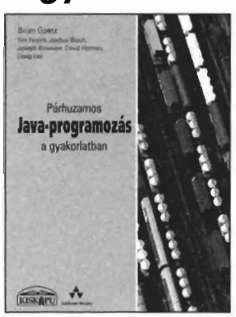
Java-fejtörők



Bolti ár: 5500 Ft
Akciós ár
a Kiskapu webboltban: 4125 Ft

Brian Goetz:

Párhuzamos Java-programozás a gyakorlatban



Bolti ár: 5980 Ft
Akciós ár a Kiskapu webboltban: 4485 Ft

Joshua Bloch:

Hatékony Java

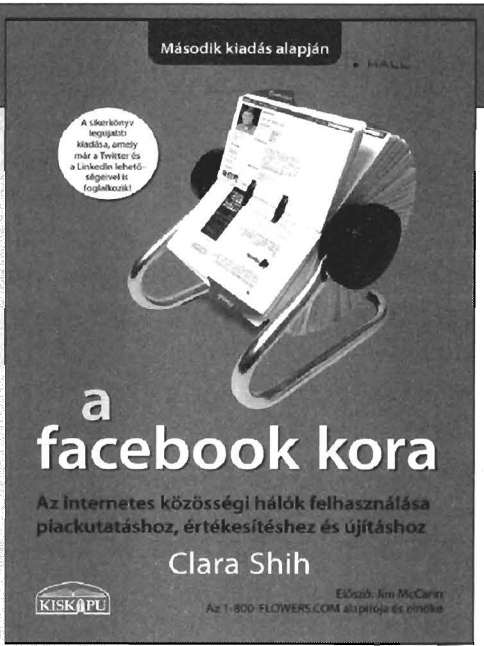


Bolti ár: 4980 Ft
Akciós ár
a Kiskapu webboltban: 3735 Ft

www.kiskapu.hu

Kiskapu Kiadó ajánlata

Clara Shih:
A Facebook kora



Marketing, értékesítés és innováció a közösségi hálók korában

Clara Shih olyan kipróbált marketingötletek oszt meg az olvasóval, melyek segítségével bármely kis- és nagyobb üzleti vállalkozás, egészségügyi és oktatási intézmény, közhasznú szervezet vagy politikai kampánystáb hatékonyabban érheti el céljait. Aknázza ki a Facebookban, a Twitterben és a LinkedInben rejlő lehetőségeket!

„Ez a könyv nem csak üzletről szól, hanem egyfajta szemléletváltásról is a társadalmat, a kultúrát és emberi mivoltunkat illetően. A jövőt senki sem látja előre, de azt tudjuk, hogy az üzlet már soha nem lesz olyan, mint azelőtt."

Bolti ár: 3900 Ft
Akciós ár
a Kiskapu webboltban: 2925 Ft

A Kiskapu Kiadó további ajánlata szakemberek számára:

Zoe Mickley Gillenwater:
Rugalmas webtervek



Bolti ár: 5500 Ft
Akciós ár
a Kiskapu webboltban: 4125 Ft

Aarron Walter:
Keresők és webhelyek



Bolti ár: 4500 Ft
Akciós ár a Kiskapu webboltban: 3375 Ft

Ryan Dewsbury:
Google Web Toolkit alkalmazások



Bolti ár: 6980 Ft
Akciós ár
a Kiskapu webboltban: 5235 Ft

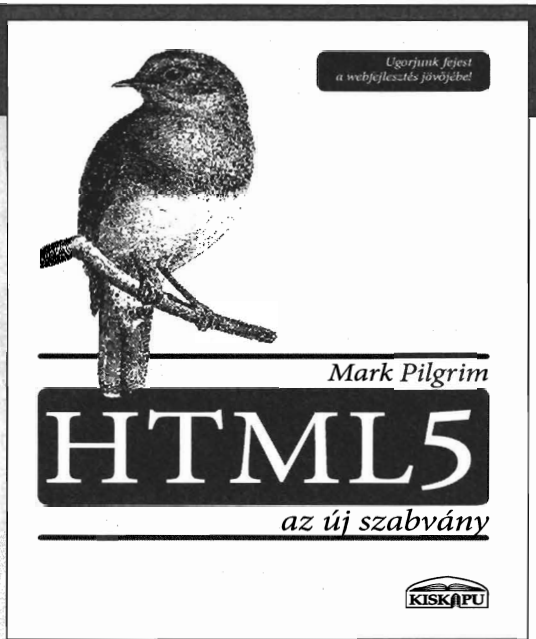
www.kiskapu.hu

Kiskapu Kiadó ajánlata

Mark Pilgrim:
HTML5 – Az új szabvány

Ha még nem ismerjük a HTML5 új szolgáltatásait, most végre elérkezett a pillanat! A kötet végigvezet minket a nyelv e változatának legfontosabb újdonságain, számtalan gyakorlati példával, kóddal, ábrával és képernyőfelvétellel. Segítségével:

- megismerkedünk a rajzvászonnal – a programozható kétdimenziós rajzfelülettel,
- videókat ágyazunk be a weboldalainkba külső bővítmények segítsége nélkül,
- a sütik lehetőségein messze túlmutató tartós helyi tárolást valósítunk meg,
- saját egyéni szótárakat hozunk létre a HTML5-ben mikroadatok segítségével.



Bolti ár: 2680 Ft
Akciós ár
a Kiskapu webboltban: 2010 Ft

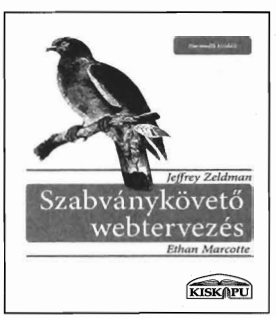
A Kiskapu Kiadó további ajánlata webmesterek számára:

Greg Rewis, Stephanie Sullivan:
CSS mesteri szinten a Dreamweaver CS3-ban



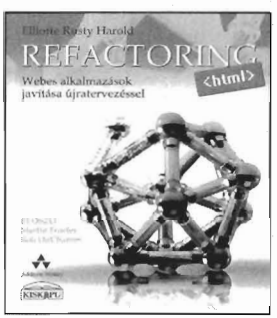
Bolti ár: 5500 Ft
Akciós ár
a Kiskapu webboltban: 4125 Ft

Jeffrey Zeldman, Ethan Marcotte:
Szabványkövető webtervezés



Bolti ár: 4980 Ft
Akciós ár a Kiskapu webboltban: 3735 Ft

Eliotte Rusty Harold:
Refactoring HTML



Bolti ár: 5500 Ft
Akciós ár
a Kiskapu webboltban: 4125 Ft

www.kiskapu.hu

Kiskapu Kiadó ajánlata

Jesse Feiler:

Tanuljuk meg a Drupal használatát 24 óra alatt

A már jól ismert 24 óra sorozat legújabb kötetében az egymásra épülő leckék közérthető nyelven vezetnek be a Drupal alkalmazásába, hogy hatékonyan és egyszerűen hozhassunk létre és tarthassunk fenn webhelyeket. Segítségével elsajátíthatjuk:

- hogyan hozhatunk létre webhelyvázakat és honlapokat automatikusan
- hogyan adhatunk szövegeket, képeket, hivatkozásokat a webhelyhez
- hogyan tehetjük biztonságossá a Drupal-webhelyeket
- hogyan vonhatjuk be a felhasználókat szavazólapok, webnaplók, fórumok és hírlevelek segítségével



Bolti ár: 6980 Ft
Akciós ár a Kiskapu webboltban: 5235 Ft

A Kiskapu Kiadó további ajánlata webprogramozók számára>

Barrie M. Noth:

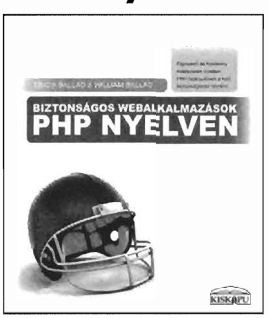
Joomla! 1.5 felhasználói kézikönyv



Bolti ár: 6980 Ft
Akciós ár a Kiskapu webboltban: 5235 Ft

William Ballad, Tricia Ballad:

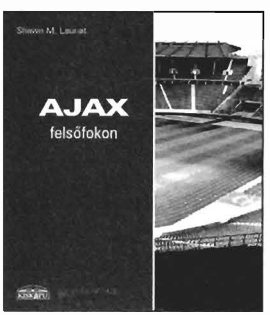
Biztonságos webalkalmazások PHP nyelven



Bolti ár: 4500 Ft
Akciós ár a Kiskapu webboltban: 3375 Ft

Shawn M. Lauriat:

Ajax felsőfokon



Bolti ár: 4980 Ft
Akciós ár a Kiskapu webboltban: 3735 Ft

www.kiskapu.hu